MASTER'S THESIS

Finding Chinks in the Armour

Software Vulnerability Prediction using Deep Learning on Graph Representations of Source Code

Elema, B.W.A (Bart)

Award date:
2020

Finding Chinks in the Armour

Software Vulnerability Prediction using Deep Learning on Graph Representations of Source Code

B.W.A. Elema

Student:
Date: 27/03/2020

Open Universiteit
www.ou.nl

FINDING CHINKS IN THE ARMOUR

SOFTWARE VULNERABILITY PREDICTION USING DEEP LEARNING ON GRAPH REPRESENTATIONS OF SOURCE CODE

by

B.W.A. Elema

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University of the Netherlands, Faculty of Management, Science & Technology
Master's Programme in Software Engineering
to be defended publicly on Friday, 27 March 2020 at 13:00.

Student ID number:		
Course code:	IM9906	
Graduation committee:	Prof. dr. M. C. J. D. van Eekelen (Chair),	Open University
	Dr. A. J. Hommersom (Supervisor),	Open University
	Dr. ir. H. P. E. Vranken (Supervisor),	Open University

An electronic version of this thesis is available at http://dspace.ou.nl/.

– ★ –

For
Alex & Marc

– ★ –

ACKNOWLEDGEMENTS

In pursuing a master's degree I have encountered more hurdles than can be summarised in this section. I have been very fortunate to have been able to count on family, friends, teachers, and colleagues to offer me a helping hand or welcome distraction during this, sometimes trying, period. In this light, I would like to thank my supervisors Dr. A.J. Hommersom and Dr. ir. Harald Vranken. Your excellent advice, suggestions and comments have kept me on track during the final stages of this study. I would also like to express my eternal gratitude to Marieke, my wife-to-be, for standing by my side during the difficult times as well as the magical ones. I know I have spent more than a reasonable amount of time staring seemingly blankly at pages of incomprehensible text scrolling across a laptop screen when there was plenty of other work to be done around the house. I do however vow to make it up to you and Marc for the rest of our lives.

Bart

CONTENTS

List of Figures

List of Tables

SUMMARY

A vulnerability could be compared to a weak link in a chain, exploiting this weakness will affect the integrity of the entire chain. As modern day software becomes more and more interlinked, software systems could be considered more akin to a chain mail than a single chain – with a single vulnerability having a possible devastating effect on the 'armour' of a software system.

With software playing an ever increasing part in daily life, the need to prevent vulnerabilities is also unrelenting. This is the reason why research into detecting and preventing vulnerabilities remains a popular subject.

As PHP is one of the most prevalent programming languages, and notoriously vulnerable to 'injection' – an attacker introducing malicious instructions into the program flow to detrimental effect on the integrity of the system – we decide to focus our research on detecting these specific types of vulnerabilities.

We base our research largely on the research on graph representations of software by Yamaguchi, Lottmann, and Rieck. [1] The code property graphs defined by Yamaguchi, Golde, Arp, *et al.* combine the expressiveness of abstract syntax trees, control flow, and program dependence graphs to capture syntactic, semantic and structural properties of source code. We propose a novel variation on these code property graphs, from which we extract paths based on edges within these graphs. These path based data points retain the syntactic, semantic and structural expressiveness while discarding superfluous elements.

We build a data set out of samples from the SAMATE PHP test suite pertaining to SQL-injection, Cross site-scripting, OS Command injection and Open Redirect vulnerabilities. We split these data sets in a training, testing and validation data set for each of these vulnerability categories with which we will train a densely connected feed-forward neural network to predict the presence of each of these vulnerabilities. Using a grid search we determine the most effective configuration for the individual deep learning vulnerability classifiers.

In comparing our classifiers to other open source approaches to vulnerability prediction, using several tools (WIRECAML, Pixy and RIPS), we find that our novel approach yields classifiers which generally match or surpass the performance of these methods. Moreover it shows the potential of combining graph representations of source code with machine learning classifiers.

Our tooling, including our pre-trained models, will be made available under the MIT licence.

1

INTRODUCTION

Software systems are becoming a more and more integral part of our lives, with any thinkable device ranging from your smartphone, your refrigerator, and billions of other devices getting connected through software and the internet. These software systems are designed to make things better or easier for their human owners and operators, allowing for food to be ordered by addressing your digital assistant or unlocking incredible computing power to anyone who requires it at the click of a button.

Many interactive software systems today are unfortunately not impervious to malicious use. Specially crafted input can be put to detrimental use on the integrity of carefully designed software. As a result, control flows might be altered, data might be tampered with or privileges might be modified to make software divert from the specification which the author had in mind. Where an isolated flaw in a stand-alone system might be allowable, an individual flawed system, when connected to other systems, forms a possible attack vector for a would-be attacker with sufficient access.

There are many attack paths an attacker could choose in an effort to gain access to a target's IT infrastructure. [3] The world's most extensive network, the Internet, has proven to be one of the most popular of these stepping stones for attackers into a specific target network. Websites are the best-known services offered on the internet, used by most if not all medium sized to large companies as a way to present their business to the world. Their prevalence makes for a interesting target for attackers. By gaining access to a target system via their web-server, an attacker could potentially escalate his privileges and move laterally through the victim's network connected to this server until he completes his objective (e.g. information theft, website defacement and using the website to infect visitors). [4]

Detecting flaws before they can be exploited by would be attackers is a central interest for the cyber security market which has grown immensely in the past decade. In 2017 Gartner predicted worldwide information security spending to reach $113 billion by 2020. [5] Research by Global Market Insights indicates that the global Cyber Security market size will continue to grow to $300 billion by 2024. [6] Gartner expects detection capability tot be a key priority for the security sector in the years to come. [5] Along with the growth of the information security market, research in vulnerability detection has been on a steady incline. As a result, data related to vulnerability detection (e.g. test suites and public code repositories) is more readily available. This in turn has opened the doors for approaches to vulnerability detection such as machine-learning. A recent survey of machine-learning

techniques in vulnerability research show that these techniques have been gaining in popularity in the past years. Academia is following this trend as well. Where, in the past, the focus had mainly been on using formal methods to prove software to be correct, more recently researchers are looking at real-world vulnerability detection. [7][1]

1.1. DEEP LEARNING VULNERABILITY PREDICTION WITH GRAPHS

We strongly believe that the role which machine learning can play in vulnerability detection will become more prevalent in the coming years. It is by this reasoning that we have chosen to look into a research question which would combine machine learning, or rather deep learning, with vulnerability detection. Furthermore we focus on the most popular web programming language at this moment: PHP. [8] Our research will present a method for vulnerability detection by answering our central research question: *"How can we effectively predict the presence of injection vulnerabilities using features learned from graph representations of source code?"*. We propose a novel method based on the notion that all software vulnerabilities originate from source code. The source code will contain indicators to the vulnerability in either syntax, structure or semantics. We build on research which abstractly represents the syntax, structure and semantics of the source code as a Code Property Graph. [1] By training a deep neural classifier to identify and recognise telltale signs of vulnerable code in these graphs aim to predict the presence of vulnerabilities in PHP-source code with relatively high accuracy and precision.

1.2. THESIS OUTLINE

In chapter 2 we discuss some preliminary concepts used throughout the thesis such as our definition of software vulnerabilities, specifically injection vulnerabilities, and existing research in the field of vulnerability detection. This chapter also explains the various graph representations used to abstract from source code. We end this chapter with a discussion on machine and deep learning. Chapter 3 describes our research design in a number of separate phases. Chapter 4 will go into how we build, optimize and evaluate our model and chapter 5 continues to compare these results with results of other open source tools. Chapter 6 concludes this thesis with summing up our research contribution, listing some limitations to our work and discussing related and future work.

2

PRELIMINARIES

This chapter touches upon a number of preliminary concepts and technologies used in our research. In particular the definition of software vulnerabilities and the analysis of software vulnerabilities as they pertain to our research, which will be covered in section 2.1 and 2.2. In section 2.3 we explore basic concepts in machine learning, which we will build upon in section 2.4 on deep learning.

2.1. SOFTWARE VULNERABILITIES

A vulnerability is a "*flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy*". [9] A vulnerability could be compared to a weak link in a chain, exploiting this weakness will affect the integrity of the entire chain. As modern day software becomes more and more interlinked, software systems could be considered more akin to a chain mail than a single chain – with a single vulnerability having a possible devastating effect on the 'armour' of an interconnected software system.

The National Institute of Standards and Technology (NIST), part of the U.S. department of Commerce, aims to index and publicize all known software vulnerabilities in the National Vulnerability Database (NVD) in order to enable users, developers and security experts to take appropriate action should their software turn out vulnerable. Each vulnerability entered into the NVD is assigned their own Common Vulnerability and Exposures (CVE) number and are placed in one of many Common Weakness Enumeration (CWE) categories.

In 2019 alone, NIST has assigned over *15,500* CVEs relating to software used around the globe on a daily basis.[10] A relatively large portion of these reported vulnerabilities can be classified as *injection* type vulnerabilities.

2.1.1. INJECTION VULNERABILITIES

An injection type vulnerability enables a would-be attacker to relay specially crafted data to a part of a system which will process it. The nature of this data can vary from textual input entered into a web form to manipulating a script to use malicious files in its control flow. Some well known examples of injection-type vulnerabilities are *cross-site scripting* and *SQL injection* which make up a large part (roughly 15%) of the NIST National Vulnerability Database. Another, less predominant injection vulnerability is *OS command injec-*

tion. In total the NVD defines five types of injection vulnerabilities. [11] We will describe the most common of these vulnerability types in more detail below.

CROSS-SITE SCRIPTING

OWASP, one of the best known online security communities, defines Cross-site scripting (XSS) as follows:

> *Cross-site scripting attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.* [12]

SQL INJECTION

OWASP defines SQL Injection (SQLi) as follows:

> *A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL command.* [13]

Our running example in listing 1 contains an example of a SQL-injection vulnerability. It is an abbreviated version of a PHP file, which contains an SQL-injection vulnerability which originates in line 2, where tainted data is introduced into the system. The data is subsequently not sanitised correctly (line 2) before being added to the intended query (line 4). The tainted data is subsequently used to query a database in line 8, completing the chain of instructions which make up this vulnerability.

OS COMMAND INJECTION

OWASP defines (OS) Command injection as follows:

> *Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application. Command injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell. In this attack, the attacker-supplied operating system commands are usually executed with the privileges of the vulnerable application. Command injection attacks are possible largely due to insufficient input validation.* [14]

Listing 1 Sample №167182, *CWE_89__POST__func_mysql_real_escape_string__multiple_AS-interpretation.php* from the SAMATE test suite (abbreviated version, appendix A contains the full listing)

```php
<?php
$tainted = $_POST['UserData'];
$tainted = mysql_real_escape_string($tainted);
$query = "SELECT Trim(a.FirstName) & ' ' & Trim(a.LastName) AS employee_name,
 ↪  a.city, a.street & (' ' +a.housenum) AS address FROM Employees AS a
 ↪  WHERE a.supervisor= $tainted ";
$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password'); //
 ↪  Connection to the database (address, user, password)
mysql_select_db('dbname');
echo "query : ". $query ."<br /><br />";
$res = mysql_query($query); //execution
while($data = mysql_fetch_array($res)){
print_r($data);
echo "<br />";
}
mysql_close($conn);
?>
```

2.1.2. RELATED VULNERABILITY TYPES

Although the NVD only defines five types of injection vulnerabilities, many more vulnerabilities related to injection can be found. [11] If an attacker can use a different avenue of approach with which to introduce tainted data in the system, for instance by modifying http parameters, this would also constitute a risk. An example of one of these types of vulnerabilities which we consider to be part of the injection type is Open Redirect.

OPEN REDIRECT

MITRE, a well known software security organization, defines Open Redirect as follows:

> *An http parameter may contain a URL value and could cause the web application to redirect the request to the specified URL. By modifying the URL value to a malicious site, an attacker may successfully launch a phishing scam and steal user credentials. Because the server name in the modified link is identical to the original site, phishing attempts have a more trustworthy appearance.* [15]

2.2. VULNERABILITY RESEARCH

As mentioned, a single vulnerability can result in failure of entire software systems. It is therefor of the utmost importance to prevent and detect vulnerabilities in software before systems are exposed to the outside world and possible attackers. Extensive research has been done in the past into how to prevent vulnerabilities in new systems, and detect vulnerabilities in existing systems.

Ghaffarian and Shahriari have done a comprehensive study on different approaches to vulnerability research. Their findings categorize three general approaches on software analysis (static, dynamic and hybrid analysis). [7]

- **Static Analysis**
  Static analysis methods are aimed at analyzing software without executing it. Most

often this is done by analyzing source code (static source code analysis). Two techniques often used during static analysis are data flow and control flow analysis, which use a graph representation of the program as the basis for the analysis. Using these types of analysis, these techniques leverage both a syntactical as a semantic view of the program under inspection. [7]

- **Dynamic Analysis**
  In dynamic analysis, in contrast to static analysis, the program *is* executed. By monitoring the program behavior and outputs while varying the input, this method is capable of drawing conclusions on the validity of the program based on the conformity of the displayed behavior to the specified behavior. Unit, integration and system testing, as well as penetration testing all fall into the category of dynamic analysis. [7]

- **Hybrid Analysis**
  Hybrid analysis uses the insight gained in static analysis to improve dynamic analysis methods. [7]

Static analysis is particularly useful in vulnerability research as it considers all the possible execution paths of the system. In the next section we will zoom into a specific type of static analysis where we represent source code using a number of directed graphs.

2.2.1. STATIC ANALYSIS USING GRAPHS

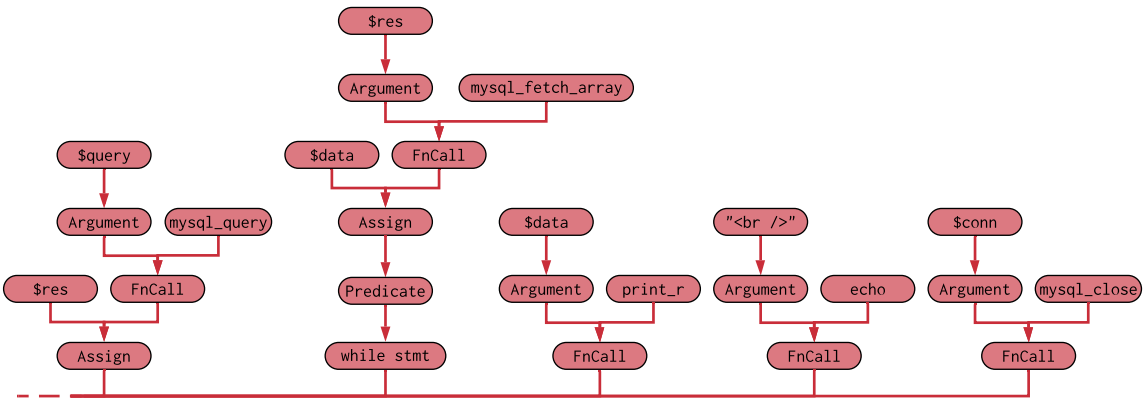
In their paper *"Modeling and discovering vulnerabilities with code property graphs"*, Yamaguchi, Golde, Arp, *et al.* introduce the notion of source code represented as (directed) graphs. These graphs contain identifiable properties indicating vulnerabilities in the represented source code. Yamaguchi, Golde, Arp, *et al.* use three different types of graphs, the AST, CPG and PDG, as building blocks to create a so called *Code Property Graph.* [2]

AST The abstract synax tree (AST) represents the basic elements (e.g. statements, variables, conditions) in a tree structure which can be represented as a property graph as defined above. While the AST purposefully captures structure, it does not capture control flow or dependencies (see figure 2.1(a)).

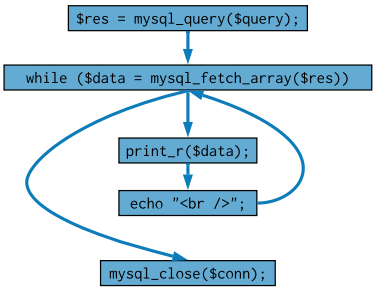
CFG The control flow graph (CFG) captures the ordering of the statements in the source code. This includes any alternative paths or loops which might be present in the source code. Furthermore the CFG might capture some of the basic properties of the structure, it does not, however, capture any data dependencies (see figure 2.1(b)).

PDG The final graph used by Yamaguchi, Golde, Arp, *et al.*, is the program dependence graph (PDG) which aims to capture dependencies in source code both in data as in control dependencies. In this sense it has some overlap with the CFG (see figure 2.1(c)).

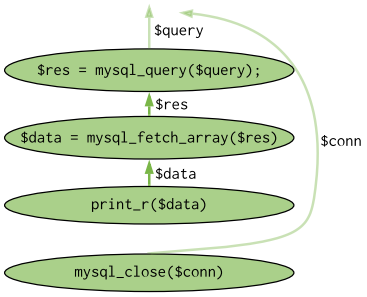
Kronjee, in his master's thesis titled *"Discovering vulnerabilities using data-flow analysis and machine learning."* proposes a method to use properties extracted from control flow graphs and abstract syntax trees for vulnerability detection in PHP applications when applying machine learning techniques. In his thesis he shows that his approach outperforms four existing tools which were built for this same purpose.



(a) AST



(b) CFG



(c) PDG

Figure 2.1: An AST, CFG and PDG representation of lines 8-13 of listing 1

Instead of using these graphs separately, Yamaguchi, Golde, Arp, *et al.* propose to combine these graphs in so-called Code Property Graphs (CPGs). These CPGs strive to capture structure, control flow and dependencies of source code. [2]

CODE PROPERTY GRAPHS

Code Property Graphs capture structure, control flow and dependencies of source code. [2] Yamaguchi, Golde, Arp, *et al.* build on the concept of a property graph for their definition of a *code* property graph.

Definition 2.2.1. Property graph is defined as $G = (V, E, \lambda, \mu)$, a directed, edge-labeled, attributed multigraph, where V is a set of nodes, E is a set of directed edges, and $\lambda : E \to \Sigma$ is an edge labeling function assigning a label from the alphabet Σ to each edge. Properties can be assigned to edges and nodes by the function $\mu : (V \cup E) \times K \to S$ where K is a set of property keys and S the set of property values. [2]

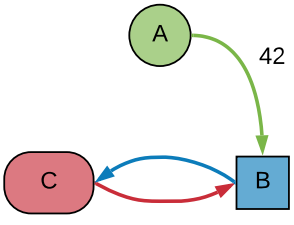


Figure 2.2: Example of a property graph with three nodes and three edges

A property graph can consist of multiple nodes which may be connected by multiple edges. Figure 2.2 illustrates a simple property graph consisting of three nodes (V =

$\{A, B, C\}$), connected by three edges. The edge labeling in this case, could be based on color that is $\Sigma = \{red, green, blue\}$ (e.g. $\{A, B\} \rightarrow green$). If we keep track of the color, text and weight of nodes and/or edges then we can define $K = \{color, text, weight\}$. Examples of key-value pairs from figure 2.2 would be $\{weight \rightarrow 42\}$ for the green edge and $\{color \rightarrow red, text \rightarrow "C"\}$ for the red node.

Yamaguchi, Golde, Arp, *et al.* define a *code* property graph as a property graph constructed by combining properties from the AST, CFG and PDG of a single code snippet into a single graph. [2]

Definition 2.2.2. Code property graph is defined as $G = (V, E, \lambda, \mu)$ with, $V = V_A$, $E = E_A \cup E_C \cup E_P$, Σ is one of $AST, CFG\ or\ PDG$ and $\mu = \mu_A \cup \mu_P$. Where the A, C and P represent the AST, CFG and PDG respectively.[2]

By traversing these code property graphs and looking for specific predefined patterns, Yamaguchi, Golde, Arp, *et al.* leverage the CPG to detect vulnerabilities. In their research they specifically looks for paths representing buffer overflow vulnerabilities, memory disclosures, memory mapping and zero byte allocations. Using these graph representations, the authors of "Modeling and discovering vulnerabilities with code property graphs" have managed to identify 18 formerly unknown vulnerabilities in the Linux kernel. [2]

Backes, Rieck, Skoruppa, *et al.*, in their paper *"Efficient and Flexible Discovery of PHP Application Vulnerabilities"* apply this same approach in detecting vulnerabilities in PHP-source code. The authors show the efficacy and scalability of the approach, applying it to a fairly large codebase of 1854 popular open-source projects. [17]

2.3. MACHINE LEARNING

Machine learning (ML) techniques have been gaining popularity with the availability of large amounts of data and a growing availability of storage capacity and processing power. Machine learning is based on statistical models built for a specific task which infer patterns from properties of data points rather than explicitly be programmed how to interpret this data. In machine learning these properties of data points are referred to as *features*.

As Ghaffarian and Shahriari note, the application of ML in vulnerability detection or prediction is increasing. [7] This section dives into the basic concepts in machine learning.

2.3.1. REINFORCEMENT, UNSUPERVISED AND SUPERVISED LEARNING

In ML, data is represented as features; measurable properties or characteristics of the individual samples. Machine learning solutions strive to predict the value, or label of a given sample by evaluating these features. A ML-model can *'learn'* how to combine and weigh each of these features using a variety of statistical models to predict these labels. Training ML-models is generally done using one of two categories of approaches: supervised and unsupervised learning.

- **Unsupervised learning**
  Unsupervised learning is aimed at grouping data according to commonalities in each sample without or with minimal human intervention. [18, pp. 104-110] An unsupervised learning technique often applied is clustering. Clustering or grouping could be applied when the training data does not have explicit labeling, but the expected

outcome is a label based on similarity to other samples i.e. the model infers the presence of distinct relations in the data without indicating on which features it should base this categorization. Clustering is illustrated in 2.3(c) where two clusters can be discerned from the data by a ML-model.

- **Supervised learning**
  Supervised learning utilizes a labeled training set to define the categories in which each sample falls. This labeling is used in order to efficiently train the machine learning model. [18, pp. 104-110]

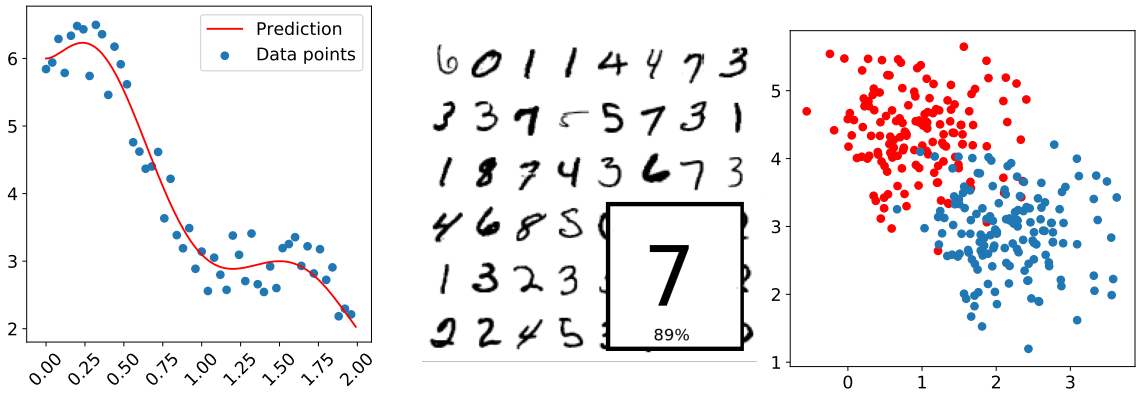
A third category of machine learning exists which is *reinforcement learning*. Reinforcement learning trains an algorithm to react to an environment by providing feedback to choices by the algorithm. The agent providing the feedback will however not provide the correct answer. This means that choices made in earlier steps will influence future outcomes. As such it significantly differs from the former categories. [19]

Each of these approaches are chosen based on both the available data for training and the definition of the problem.

REGRESSION AND CLASSIFICATION

Supervised learning has a number of different implementations which each have their merits based on the problem one would like to solve and the data at hand. After the data has been thoroughly analysed and understood, the next choice in implementing a machine learning solution is determining the type of problem. We will focus on two types of problems which supervised learning can solve.

- **Regression problems**
  Regression problems are problems where the model is tasked to find as an outcome, a continuous value based on the values in the training data. An example of a regression problem would be predicting a persons income based on their age and education. Figure 2.3(b) illustrates regression by inferring a predicted value Y for every X from the data points present; [18, pp. 100-101]



(a) Regression (b) Classification (c) Clustering

Figure 2.3: Examples of different types of typical ML problems.

- **Classification problems**
  Classification problems have as an intended outcome a label based on the data with which the model has been trained. One of the best known examples is the MNIST handwritten digit classification problem. In this problem a large data set of images of handwritten numerals from 0-9 are provided. The object is to predict which number a given image from this data set represents. Figure 2.3(b) shows a part of the (labeled) MNIST data set. Classification problems typically have distinct categories or labels, which are also present in the training data; [18, pp. 100-101]

2.3.2. PERFORMANCE METRICS FOR CLASSIFIERS

In order to evaluate the performance of a given classification model, a number of metrics can be used.

Generally these metrics do not refer to (computation) time or complexity, but rather to the ability of the model to correctly predict outcomes.

When dealing with classification problems, the number of *True Positives* (TP), *True Negatives*, (TN), *False Positives* (FP) and *False Negatives* (FN) are leading concepts in defining these metrics. Generally speaking, classifieres primarily output a real value which has to meet a certain *classification threshol*d to determine a definitive label. *True Positives* and *True Negatives* refer to correctly classified samples, be they a positive sample classified as positive or a negative sample classified as negative. *False Positives* and *False Negatives* refer to incorrectly classified samples i.e. positive samples classified as negative or negative samples classified as positive. [20] We can combine these values in a so called confusion matrix, shown in table 2.1.

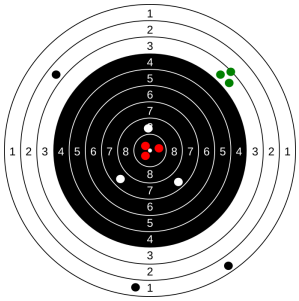


Figure 2.4: This bullseye indicates the difference between *accuracy* and *precision*. The red and green dots are grouped in small groups, they indicate *precise* scores. The red and white dots are close to the bullseye and indicate *accurate* scores. The black dots are nether *precise* nor *accurate*. Only the red dots are both *accurate* and *precise*.

Table 2.1: Example of a confusion matrix with two classes: *A* and *B*

	Classified A	**Classified B**	
Actual A	50 (TP)	10 (FN)	60 (Total actual class A)
Actual B	20 (FP)	100 (TN)	120 (Total actual class B)
	70 (Total classified as A)	110 (Total classified as B)	180 (Total A+B)

DERIVED METRICS

From the number of true and false positives and negatives we are able to derive a number of secondary metrics:

- **False Positive Rate (FPR)**
  The FPR is the negative variant to recall. It gives an indication of what percentage

of the *negative* samples it will reliably classify correctly. FPR is defined as $FPR = \frac{FP}{FP+TN}$. [20]

- **Accuracy**
  Accuracy, as indicated in fig. 2.4, is the rate of correctly classified samples. Note that this includes both positive and negative (TP and TN) samples. This rate is calculated as $Accuracy = \frac{TP+TN}{TP+FP+FN+TN}$. [21]

- **Precision**
  Precision indicates the reliability of the prediction i.e. the rate of correctly classified positive samples. $Precision = \frac{TP}{TP+FP}$ [22]

- **Recall**
  Recall is related to to the precision, the difference being that recall indicates the rate of positive samples classified correctly. This means that this score gives an indication of what percentage of the *positive* samples it will reliably classify correctly. We calculate recall as $Recall = \frac{TP}{TP+FN}$. [22]

- **F_1 Score**
  In practice, machine learning solutions often have to make a trade off between precision and recall; Improving precision will often negatively influence recall and vice versa. This trade off is quantified by the F_1-score or F_1-measure:
  $F_1 = 2 * \frac{Recall * Precision}{Recall + Precision}$. [23]

- **ROC-AUC**
  The ROC curve or *receiver operating characteristic* curve is a graph which depicts relative trade-offs between benefits (true positives) and costs (false positives). [24] It plots the relationship between recall and the false positive rate at all classification thresholds. By taking the *area under* the ROC *curve* (AUC) as a measure we obtain the ROC-AUC. AUC provides an aggregate measure of performance across all possible classification thresholds. [25] The downside of the ROC-AUC is that it does not always provide a meaningful metric with heavily skewed data sets (i.e. one class is heavily underrepresented in the data set). [26]

- **PR-curve**
  The PR curve or *precision-recall* curve is a graph which depicts the performance of a classification model. It shows the relationship between precision and recall for every possible classification threshold. Every point on the PRC represents the precision and the recall for every chosen threshold. By taking the *area under* the PR *curve* (AUC) as a measure we obtain the PR-AUC. [26]

Precision and recall (and the derived F_1-score) are by definition based on the ratio of correct positive predictions to the *total predicted positives* (precision) or to the *total positives examples* (recall). Switching the viewpoint on these classes (positive → negative and negative → positive) will possibly give more insight into the performance of a classifier on negative samples. [27] The average scores for each of these viewpoints can be represented in two

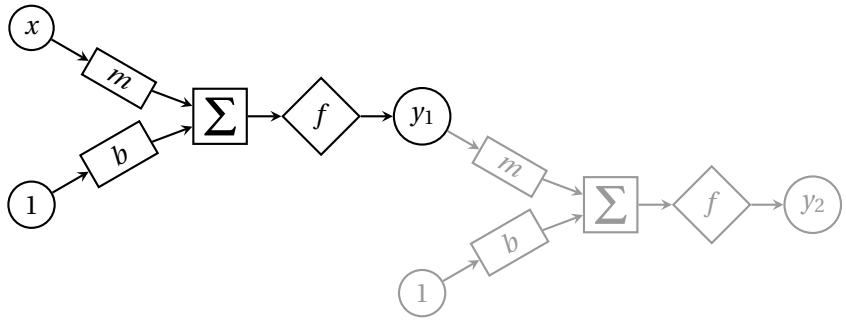


Figure 2.5: Two perceptrons in tandem as part of an artificial neural network. With x representing an input and b and m representing bias and weight. Function f represents the activation function.

ways, *macro* and *weighted* averages. Weighted averages incorporate the distribution of the different classes by adding more weight to the most predominant class. Macro averages do not incorporate the difference in distribution and take a naive average of the positive and negative value. [28]

2.4. DEEP LEARNING

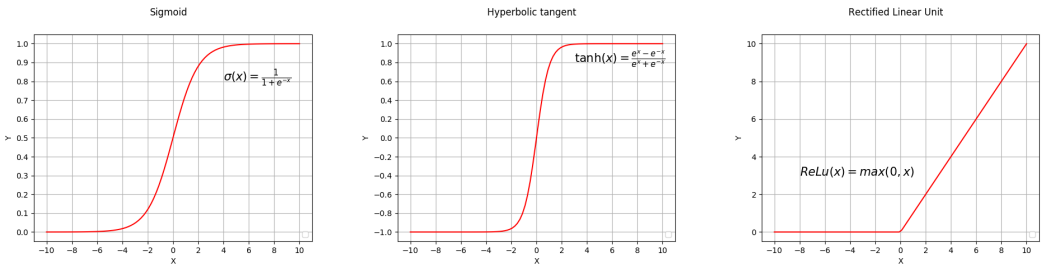
Deep learning is a subcategory of ML which, in contrast to basic ML, allows a model to learn features from (unstructured) data, where traditional ML needs these features to be provided. This means that deep learning, using a so called (artificial) neural network, does not require structured data as is the case with traditional machine learning. This trait allows neural networks to successfully tackle computationally difficult problems such as image and voice recognition, and natural language processing.
This section will describe the workings of a neural network by zooming into three basic elements – the architecture, training and (hyper)parameters.

2.4.1. ARCHITECTURE

Each neural network is built up out of a number of different building blocks, each with a distinct function in processing or learning behavior in the model. The type of building blocks and the way they are arranged and interconnected is referred to the architecture or topology of a neural network.
Neural networks can be tailored to a specific problem by varying the architecture. We will focus on describing a generic densely connected (feed forward) neural network. There are many variations to this type of neural network (e.g auto-encoders, long/short term memory models, recurrent neural networks and gated recurrent neural networks) which are outside the scope of this thesis.

PERCEPTRON

The basic building block of the artificial neural network is the equation $y = f(mx+b)$ which can be represented as a perceptron. Figure 2.5 illustrates the basic outline of two perceptrons, where the first perceptron's output feeds into the next perceptron's input. Generally speaking, a single perceptron will take in a weighted input and add a certain bias before feeding that outcome into an activation function. Without the activation function $y = mx+b$ would be no more than a linear function. By adding an activation function, i.e. a threshold value which $mx+b$ needs to reach, this linear trait is removed. The result is that



(a) $\sigma(x) = \frac{1}{1+e^{-x}}$ (b) $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ (c) $ReLu(x) = max(0, x)$

Figure 2.6: Plot of the (a) sigmoid, (b) tanh and (c) ReLu activation functions

by manipulating the *weight* for each perceptron (or unit) we will be able to influence which units are activated and which are not for a given input. [29, p. 65] The activation function is generally chosen to be some type of non linear equation such as a sigmoid ($\sigma(x)$) or hyperbolic tangent ($tanh(x)$). In recent years however, the *Rectified Linear Unit* ($ReLu(x)$) has been the activation function of choice to use in hidden layers as it has several traits superior to the former two, figure 2.6 illustrates these activation functions. [30]
The output unit does typically not use a ReLu activation function. As its task is to output a predicted value and a confidence in this prediction, the output unit generally uses either linear activation (for linear regression problems), sigmoid for binary classification and *softmax* when predicting multiple classes. [30][31, p. 71][18, p. 184-187] The specifics of the softmax activation function lie outside the scope of this thesis.

LAYERS

The first stage of building a neural network is placing a number of perceptrons in parallel to form a layer. The exact number of perceptrons is determined by the number of data points in the input data. Stacking multiple layers builds a multilayer perceptron or *neural network*. A typical neural network consists of at least three types of layers; The first is the input layer, which, as the name suggests, is the primary data interface for the model. Connected to the input layer are generally one or more hidden layers before reaching an output layer. [29] With each added layer, the network is able to extract progressively abstract characteristics or features from the data. [29]

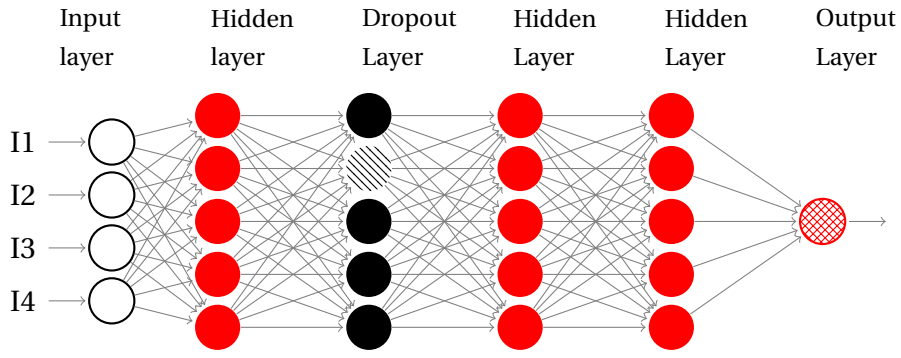


Figure 2.7: An example of a densely connected neural network featuring four hidden layers. The second hidden layer is a dropout-layer which, in this example, disables 20% of the nodes.

Figure 2.7 also features the use of a dropout layer where a certain percentage of the nodes

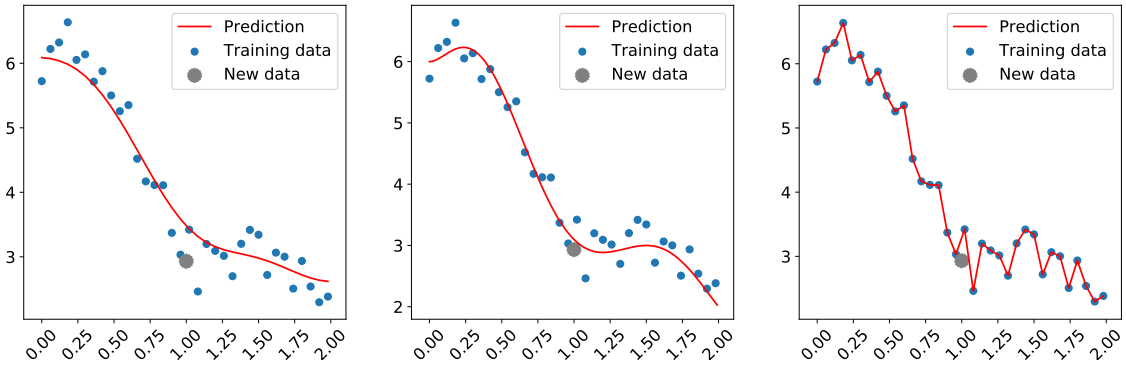
in that layer is disabled. Adding dropout makes a trained network more robust.[29, pp. 103-104]

2.4.2. TRAINING

In training a neural network we strive to adjust the way the network processes input, along its layers, to match an expected output. We can use a labeled (training) data set for this purpose. By feeding the network samples from this data set we are able to determine how the output produced by the network ($\hat{y}$) differs from the desired output or label (y). This difference is expressed as *loss*. As noted in the previous section we can manipulate the weights of perceptrons to alter their behavior. By adjusting weigths to reduce the difference between the labels of the samples and the given output (i.e. minimize *loss*) after each training cycle, we are telling the network which behavior is desired. We refer to the combination of architecture and the (trained) weights and biases of a neural network as a model or in the case of a classification model, a *classifier*. [18][30]

BACKPROPAGATION

This manipulation of the weights is however not straightforward. The adjustments need to be propagated back through the network to allow every hidden layer from output to input to be adjusted. This process is called *back-propagation* or *backprop* and it defines the 'learning' ability of the network. [32][31, pp. 51-52][29, pp. 57-65]



(a) underfitting (b) correct fit (c) overfitting

Figure 2.8: Three different stages of fitting the model (labeled *Prediction*) to the training data.

OVERFITTING

Neural networks are prone to a number of unwanted effects during training. The most common being overfitting. Overfitting occurs when a model learns to recognise the training data set itself. This will mean that the model will perform extremely well on the training data set, but will be unable to correctly predict labels of unknown input. [18, pp. 110-116] This effect is demonstrated in figure 2.8 with a model underfitting the data (fig. 2.8(a)), correctly fitting the data (fig. 2.8(b)) and overfitting the data (fig. 2.8(c)). As the figure shows, the model is able to match the new data most closely to the prediction with a correct fit. This phenomenon is the primary motivation for strictly dividing testing an training data sets. A model which does not suffer from overfitting will have similar performance when

Figure 2.9: K-fold cross validation illustrated, with the (changing) validation data set in red.

used to predict labels for the training or testing data set. Adding dropout also prevents a model from recognizing individual samples and as a result it prevents overfitting.[18, pp. 258-267][33]

K-FOLD CROSS-VALIDATION

If a given data set has very few observances of a certain class the model is prone to learning to favor the other class. In order to prevent this behavior, with minimal risk of overfitting, in some cases K-fold cross validation is applied. K-fold cross-validation divides the training data in K parts, with each part consisting of roughly equal proportions of classes of data (e.g. *safe* and *unsafe*) as the proportions featured in the whole training data set. Subsequently a model is trained on $K-1$ of these parts and evaluated on the remaining part, illustrated by figure 2.9. This process is repeated K times with a different part used as evaluation each time. By training K models, and averaging the scores of these models we get a relatively accurate estimate of the performance of a given classifier. [31, p. 99][18, p. 122]

2.4.3. HYPERPARAMETERS

Hyperparameters are the parameters which control both architecture and training. Hyperparameters could be compared to knobs and dials with which to tune the model. Finding the right settings of these hyperparemeters is often a challenge as it is not always predictable which configuration is best suited for which problem, goal and input. [30]

Table 2.2: Different hyperparameters explained

Parameter	**Description**
Hidden layers	The number of hidden layers used in a model. Increasing the number of hidden layers allows the model to learn more specific behaviour. A larger number of layers will however require more memory and more time to train. [18, p. 169]
Units per layer	The number of units per (hidden) layer. By reducing the number of units per layer the model is forced to describe more data using less (weigths of) units. The intuition here is that these units will only retain those values which are most descriptive. [18, p. 429]
Dropout	The percentage of units which are 'disabled' in a given layer.

Parameter	Description
Activation function	As described in fig. 2.6 there are a number of possible activation functions per hidden layer, each presenting a different behavior. [18, pp. 174-190]
Epochs	The number of times each sample is used to train the model. This includes both a forward and backward pass through the network. [18, p. 246]
Batch size	The training data set is divided up into multiple batches which are processed consecutively. Between each batch loss is calculated and backprop is performed. The batch size is the number of samples *Batch size*, as a result determines how often a models weights are adjusted. Larger batches might take longer to train, but tend to consume a lot of memory. Models trained using smaller batches tend to train faster, but less efficient due to fluctuating gradients. [18, p. 78]
Learning rate	The learning rate describes the factor by which the weights are corrected per iteration. [18, pp. 185-186]
Optimization function	The optimization function is the function driving backprop. It feeds back outcomes through the network and varies the weights. Popular optimizers are Adaptive Moment Estimation (Adam) and Stochastic Gradient Decent (SGD), which each have their individual merits. [34]
Loss function	The loss function is the function which indicates the difference between your goal and the actual output. The loss in deep learning is used to tell whether the adjustments made to the weights of the model by the optimizer are beneficial or detrimental to the model. [18, p. 82]

2.4.4. DEEP LEARNING FOR VULNERABILITY PREDICTION

As Ghaffarian and Shahriari note, the application of ML and deep learning in vulnerability detection or prediction is increasing. [7] Li, Zou, Xu, *et al.* in their paper *"VulDeePecker: A deep learning-based system for vulnerability detection"* propose a deep learning approach (VulDeePecker) to detect vulnerabilities in source code based on semantically linked lines of (C/C++) code. [35] The authors improve on this concept in their paper *"SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities"*, where they combine two different types of source code representations (i.e. semantic and syntactic) to allow for better representation of potentially vulnerable elements in source code. They train a Long/short term memory (LSTM) deep leaning model which significantly improves on their earlier work.[36]

In *"Building program vector representations for deep learning"*, Mou, Li, Liu, *et al.* suggest using ASTs as a basis for program vector representations to use in classification of source code using deep learning. This concept is elaborated on by Alon, Zilberstein, Levy, *et al.*, where the author attempts to classify source code by vector representations of source code tokens.

We believe that capturing both semantic and syntactic details of source code is key in pre-

dicting vulnerabilities. Our approach will combine the expressive power of descriptive graphs as Yamaguchi, Lottmann, and Rieck and Kronjee have done, with deep learning, using concepts used by Mou, Li, Liu, *et al.* and Alon, Zilberstein, Levy, *et al.*

3

RESEARCH DESIGN

Our research is designed around a central research question, which we will describe in the first section of this chapter. Subsequently, in section 3.2 we will go into the method we have designed and employed in this research. The final section will elaborate on the contributions of our research.

3.1. RESEARCH QUESTIONS

This thesis describes our research in creating a deep learning based approach to detecting injection type vulnerabilities using graph representations of PHP source code. To this end we will answer the following research question:
"***How can we effectively predict the presence of injection vulnerabilities using features learned from graph representations of source code?***"
We answer this question by employing empirical research based, in part, on research done by Kronjee, Yamaguchi, Lottmann, and Rieck and Li, Zou, Xu, *et al.* [16][1][2][39][35][36] To answer this question we break it down in three sub-questions:

R.Q.1 *How can we represent Code Property Graphs so that they can be successfully exploited in a deep-learning approach?*

R.Q.2 *How can we develop a classifier, able to predict vulnerabilities based on features generated from code property graphs?*

R.Q.3 *How does the proposed method perform compared to other automated approaches to vulnerability detection (i.e. open-source tools and comparable research)?*

The notion of capturing properties of software by representing the source code as a graph presents an interesting view. Yamaguchi, Lottmann, and Rieck and Xiaomeng, Tao, Runpu, *et al.* have shown the advantages of using the code property graph as a basis for vulnerability prediction.

3.2. RESEARCH METHOD

Our general approach is visualised in figure 3.1. Generally speaking, we first extract vulnerable and safe source code samples from the publicly available SAMATE test suite. The samples in this data set are subsequently transformed to code property graphs. From these

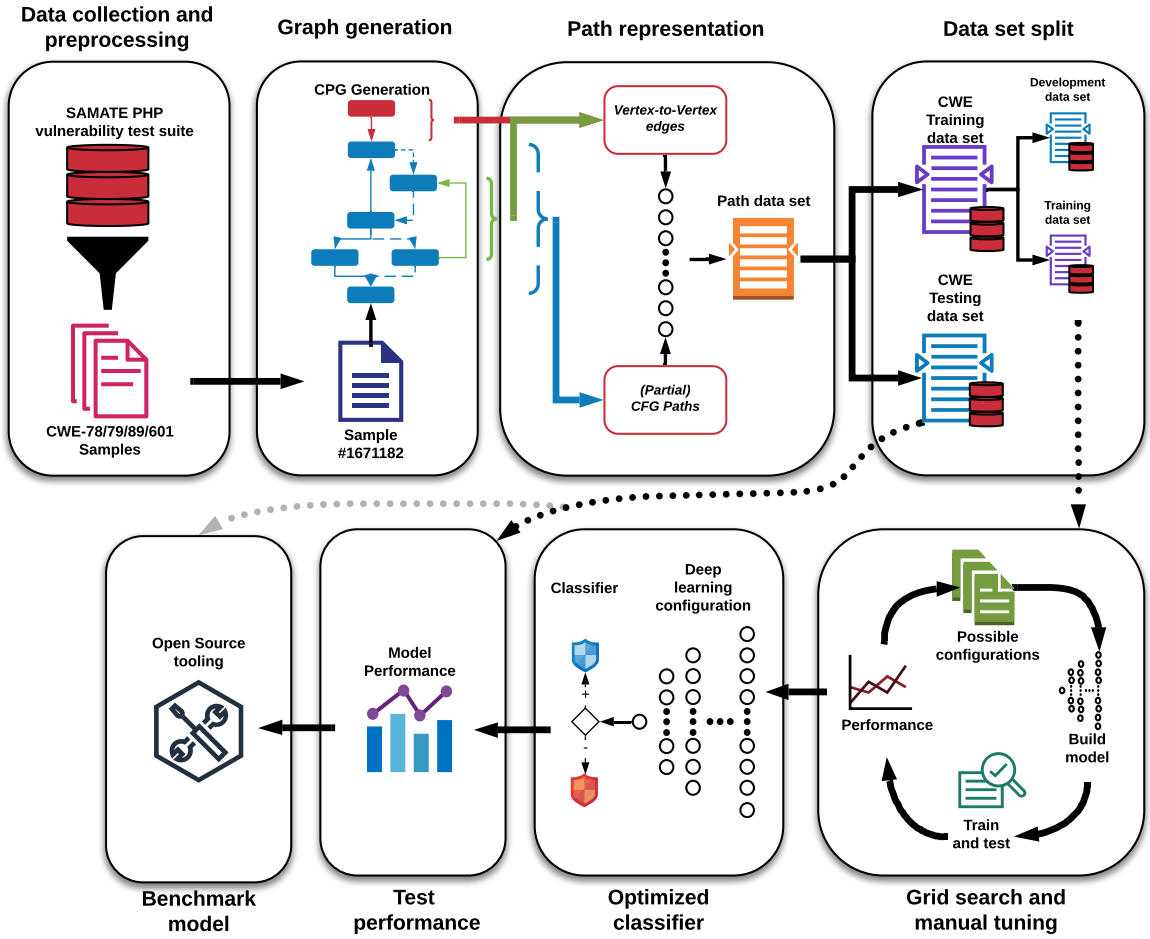


Figure 3.1: Our general approach visualised in 8 stages (Data collection and preprocessing steps, including graph generation, path extraction and data set splitting. Followed by building, tuning and optimizing a deep learning classifier which is subsequently tested and compared to other tools.)

graphs we extract path based data points and combine these in a data set. The data sets resulting from these initial stages are used to automatically train and evaluate deep learning classifiers in search of the top performing classifier for each vulnerability category in table 3.1. In the next phase we test our optimised classifiers, their performance is subsequently compared to the results of several publicly available vulnerability detection tools.
Our method varies on Yamaguchi, Lottmann, and Rieck by proposing a novel variation on the code property graph, from which we extract specific paths. We furthermore use a grid search based on a predefined set of hyperparameters to determine the most effective deep learning classifier for each vulnerability in table 3.1. In comparing our classifiers to other open source approaches to vulnerability prediction we find that our novel approach has some advantages over other methods. Moreover it shows the potential of combining deep learning with graph representations of source code.

3.2.1. PHP SOURCE CODE REPRESENTATION AS CODE PROPERTY GRAPHS

Yamaguchi, Maier, Gascon, *et al.* suggest that the Code Property Graphs described in section 2.2.1 present us with a method for successfully capturing semantics and syntax of source code. [39]
Based on this notion, we try to answer the following sub-question: *How can we represent Code Property Graphs so that they can be successfully exploited in a deep-learning approach?*

DATA SET SELECTION

The data available to train a model is a key factor in choosing the approach to solving a problem using machine, or in our case deep learning. Furthermore the quantity and quality are key factors in successfully training an accurate model. To make certain the model has enough information to adjust its weights and biases to match the required behavior, a sufficient amount of data is required. It is difficult to know beforehand how much data is sufficient, the accuracy of a model will however generally benefit from larger quantities of data.
The quality of the data is also an important factor. With a classification problem, care should be taken that all classes are distinct and that all the identified categories are represented in adequate proportions by the data. A data set is allowed to contain some 'noise' in the form of incorrectly labeled data. Noise, in moderate amounts, creates a robust model, decreasing the impact of outliers. It is therefor not necessary to inspect the labeling of each input sample individually.
Even though our approach should generally be applicable to other programming languages and vulnerabilities as well, we have chosen to primarily focus our research on PHP. The reasoning behind the choice for PHP is twofold. Firstly, PHP has held a top 10 spot on the niobe list of most popular web programming languages worldwide since 2003, furthermore PHP is used in over 75% of all server side programming. [8] Secondly, the substantial number of entries of PHP related vulnerabilities in the National Vulnerability Database indicate that applications built using PHP are prone to containing vulnerabilities. [41]

SAMATE PHP TEST SUITE

On their website the National Institute of Standards in Technology (NIST) hosts a number of reference data sets specifically for vulnerability research. One of these is the Software Assurance Reference Data set for PHP, generated by Bertrand Stivalet and Arelien Delaitre. The test suite was published by the Software Assurance Metrics And Tool Evaluation (SAMATE) project. [42] The aim of the SAMATE project is to allow automated tools to evaluate their performance in vulnerability detection against standardised data sets. The PHP test suite contains 42239 test cases labeled by vulnerability category of which 41712 pertain to injection vulnerabilities. The freely available data set comes with a manifest providing a short description on each sample and outlining the type of vulnerability.
Although the data set contains eight categories of injection vulnerabilities (i.e. CWE-78: OS Command injection, CWE79: XSS, CWE-89: SQLi, CWE-90: LDAP injection, CWE-91: XML-injection, CWE95: Eval injection, CWE-98: Remote file inclusion and CWE-601: Open redirect), we limit our research to the categories listed in table 3.1 as they comprise the vulnerabilities most often exploited. [10]

Table 3.1: Distribution of vulnerability categories for the subset of the SAMATE test suite used in our research

Category	Description	# Safe samples	# Unsafe samples	# Total
CWE-78	OS command injection	1872	624	2496
CWE-79	Cross site scripting	5728	4352	10080
CWE-89	SQL injection	8640	912	9552
CWE-601	Open redirect	2208	2592	4800
		18448	8480	26928

Each test case is stored in a separate PHP file containing a preamble in the form of a comment describing whether the file represents a safe or unsafe sample and a disclaimer (see listing 2). Secondly it contains a section of PHP code, commented if needed. The data set was generated to contain a single vulnerability per sample. The type of the vulnerability can be derived from the filename, which also includes the CWE-category. An abbreviated sample from the SAMATE data set is shown in listing 1. This sample will serve as a running example throughout this thesis.

GRAPH REPRESENTATION

Our research builds on research by Yamaguchi, Lottmann, and Rieck into using graph representations of source code for vulnerability detection. To prepare the data set for use as input for training our model, we represent the PHP samples as a code property graph much like defined in section 2.2.1.
We do alter the definition given by Yamaguchi, Golde, Arp, *et al.* slightly, reusing the labeling function for both nodes and edges, see definition 3.2.1. The code property graph (CPG), as explained in section 2.2.1, is a complete representation of the semantic and syntactical data present in the source code. The created graph consists out of all the basic syntactic elements from the AST and CFG representing nodes. The semantics such as data dependency are captured by the edges which indicate the relations between the nodes.

Definition 3.2.1. Our Code Property Graph is defined as $G = (V, E, \lambda, \mu)$, a directed, edge-labeled, multigraph where V is a set of nodes, E is a set of directed edges, and $\lambda : (V \cup E) \rightarrow \Sigma$ is a labeling function assigning a label from the alphabet Σ : (*AST*, *CFG* or *PDG*) to each node and edge and where node values are mapped by the function $\mu : V \rightarrow S$ where S is the set of node values (e.g. `If`, `Expr_assign`, `Terminal_Echo`, etc.).

Where Yamaguchi, Golde, Arp, *et al.* build the CPG by combining properties derived from the AST, CFG and PDG into a single graph, we have opted to use a tool named PHP-CFG[1] by Anthony Ferrara to generate a CPG from PHP-source code. As the name indicates, PHP-CFG is intended to generate the control flow graph. In parsing the tools output however we reconstruct the AST on which the CFG is based. At a later stage we add data and control dependencies, based on the AST-CFG combination and add them to form the final (full) CPG.
The code property graph in the full form as displayed in figure B.1 contains a number of nodes and edges very specific to a given sample. To allow our solution to apply to generalize more effectively, the generated graph requires a certain degree of abstraction. Figure B.1, for instance shows each variable, including its name as a separate node in the CPG, while we believe these variable names to be of lit-

[1] PHP-CFG by Anthony Ferrara on Github https://github.com/ircmaxell/php-cfg

tle informative value. An extremely specific example is the node labeled "`LITERAL('SELECT Trim(a.FirstName)...WHERE a.supervisor=')`", which is unique to this sample.

Training a classifier on these types of specific entities will not likely benefit its functionality. We believe that removing such sample-specific entities will allow features extracted from these graphs to be more generally applicable. We prune the graph by removing nodes representing variable names and strings which we deem to hold little expressiveness outside of the context of a single specific sample. This step removes *nearly* all of the AST nodes. As is shown in the resulting graph in fig. 3.2 we do retain nodes representing so called *superglobals* (e.g. `$_POST` and `$_GET`). Superglobals hint at data which is introduced into the system by end-users and as such could be an indicator for a possible exposed attack vector. Furthermore we retain function names as a part of the underlying function call. Function names could hint at how variables are processed in potentially vulnerable functions or if and how tainted data is sanitised. An added advantage of pruning is that the feature space (i.e. the corpus of generated features) is significantly reduced.

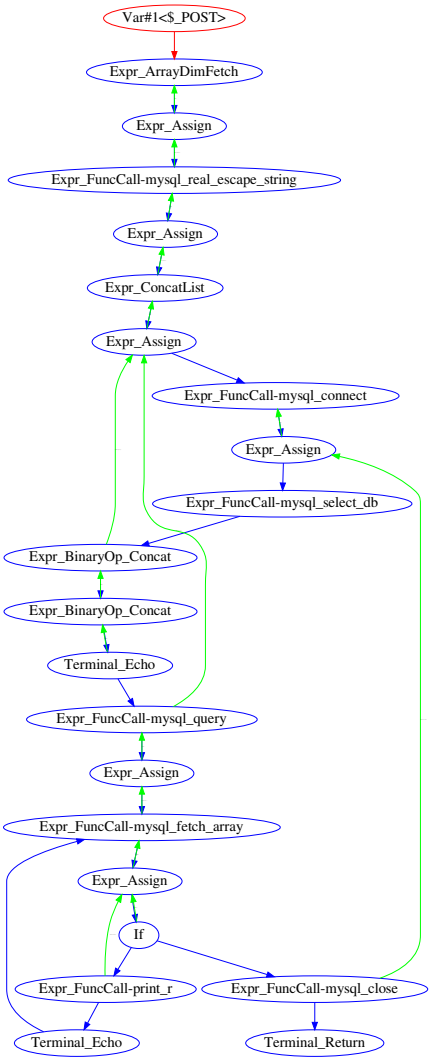


Figure 3.2: Pruned CPG representation of sample №167182. Blue vertices represent control flow dependencies, green vertices represent a data dependency and blue vertices indicate a relation in terms of the abstract syntax tree.

Definition 3.2.2. Our Pruned CPG, $G = (V, E, \lambda, \mu)$ limits the V in to $V = AST' \cap CFG'$, with AST' as the subset of AST-nodes representing Superglobals and CFG' as the full set of CFG-nodes, with function call nodes (i.e. `Expr_FuncCall`) modified to include the function name (e.g. `Expr_FuncCall-mysql_real_escape_string`).

Pruning the graph allows us to make a tradeoff between expressiveness and generalization. Retaining specific details will most likely result in more accurate classification, removing specific details will reduce precision. We believe we have found an adequate balance by using definition 3.2.2 as a guideline.
We generate 22491 graphs from the available samples.[2]

[2]Roughly 17% of the samples fail to process, this is due limitations of the underlying *php_cfg*-script which seems unable to handle a number of node types (e.g. `group use`). As the samples and the specific reasons for failure are quite diverse we assess that the loss of these samples will not significantly influence the performance of our classifiers negatively.

PATH EXTRACTION

We believe that the CPG holds all the useful elements needed to represent the vulnerable syntax and semantics of the source code. In order for a neural network to discern useful features from these structures we opt to represent the CPG by deriving path-based data points from the CPG structure.
As we would like to capture as many data points as needed, we consider every connection in the graph to be of importance i.e. every tuple of the form (V, E, V), where V represents a node and E represents a (labeled) edge of type AST, CFG or PDG. This guarantees that each node is represented, as part of these node-to-node tuples in the data set.
These node-to-node tuples however do not fully capture the order in which statements are executed in code. This ordering is especially important when evaluating whether variables have been sanitised before being passed to a possibly vulnerable function.
This is why we also consider every possible path from the first node (i.e. the first line of code) to the last (i.e. the last line of code). These full paths will undoubtedly include any sanitation statements present between introduction of a variable and using it in a function. The collection of full paths derived from a single sample offers complete coverage of all the elements, syntactical and structural, of the underlying graph. Hence we do not lose any information in this conversion.
Despite earlier measures to generalise the graphs as much as possible (see pruning, section 3.1), these full paths will be quite specific for a sample. To allow us to generalise these paths, we generate unique sub-paths.

Definition 3.2.3. These (sub) paths can be defined as $(V_i, E_i, \ldots, E_{j-1}, V_j)$, where V represents a node, E represents a (labeled) edge of type AST, CFG or PDG, and i and j represent different nodes in the sub-path. The minimal and maximal length of these sub-paths are determined by parameters (equal to $j - i$).

Examples of both categories of features are shown in table 3.2. The top four samples representing the former category of node-to-node data point and the last line representing the latter of a path-based data point.
Note that by changing the minimal and maximal sub-path length, we can influence to what extent a path is specific to a certain sample. If the *minimal* parameter is equal to the length of the longest path in the graph then the sub paths are equivalent to the full paths. If the *maximal* parameter is equal to 1 then the sub-paths are equivalent to the node-to-node tuples. Too specific (i.e. long sub-paths) will not generalise to other samples, not specific enough (i.e. very short sub-paths) will not contain enough information on the ordering of statements and as a result, will not be useful in prediction of the presence of vulnerabilities.

Table 3.2: Examples of tuples of edges used as features generated from sample №167182. The *'a','p'* and *'c'* represent the type of node or arc: *ast*, *pdg* or *cfg*.

```
a:$_POST/a/c:Expr_ArrayDimFetch
c:Expr_ArrayDimFetch/c/c:Expr_Assign
c:Expr_Assign/p/c:Expr_ArrayDimFetch
...
c:Expr_FuncCall-mysql_fetch_array/c/c:Expr_Assign/c/c:If/c/c:Expr...
```

When grouping samples according to vulnerability category and plotting the number of features generated per sample for each category (fig. 3.3) we see that on average each sample is represented by ≈56 features, which comes down to ≈1½ feature per line of code.

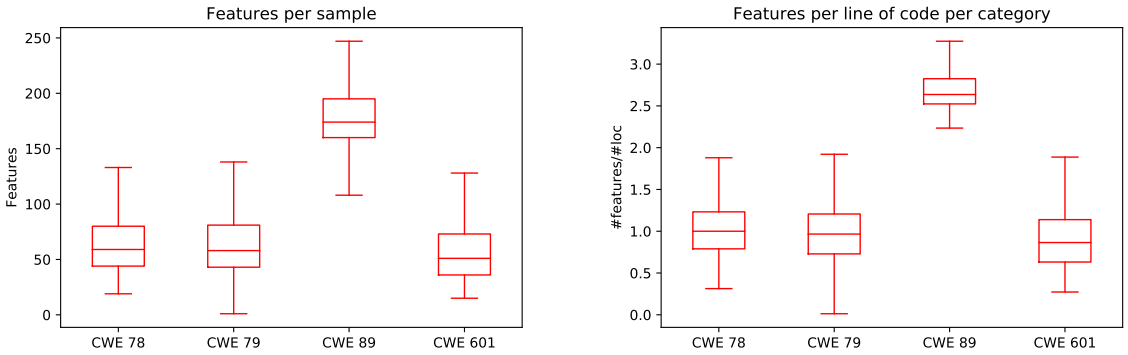


Figure 3.3: Distribution of number of features per sample and per line of code for each CWE-category

It is noteworthy that SQL-injection (CWE-89) generates more features per sample and line of source code on average, compared to other categories. A reason for this phenomenon could be that this specific category of vulnerabilities requires more source code to be expressed. More source code per sample will invariably yield a more elaborate CPG which in turn will contain more features.
The relatively large difference in features per category also indicates that it would be quite difficult to build a comprehensive model for all of the vulnerability categories which could take into account these differences. This is the primary reason why we choose to build separate classifiers for each category of vulnerabilities.

TRAINING, VALIDATION, DEVELOPMENT AND TEST DATA SETS

After having extracted edges and paths from all the Code Property Graphs we aggregate these outcomes in a separate *path data set* for each vulnerability category. This is illustrated in box 3 in figure 3.4. This leaves us with four path data sets which represent all the paths and edges from the converted graphs generated from the original selection of PHP files. Table 3.3 shows part of the path data set for SQL injection.

Table 3.3: Example of (part of) a path data set. Besides the presence of any of the generated paths and edges for each sample (represented by the columns), we also store the label as provided by the SAMATE test suite.

	`a:$_POST /a/c:Expr...`	`c:Expr _ArrayDim...`	`c:Expr _Assign/p...`	...	label
Sample №167182	1	1	1	...	unsafe
Sample №167183	1	0	0	...	safe
Sample №167184	0	0	0	...	safe
Sample №...	...	...	...	...	...

Before building our models we split the path data sets in training, development and test data sets. This is shown in figure 3.4 by the path data set, labeled 3, ultimately being split into a development data set (3a1), a training data set (3a2) and a test data set (3b). The development data set is used to evaluate the performance of the classifier during its development, the training data sets are used to train the classifier, either during development (3a2) or when training the final classifier (3a). The split ratio we use is approximately 58%, 32%, and 10% for the training (3a2), development (3a1) and test data sets respectively (3b).

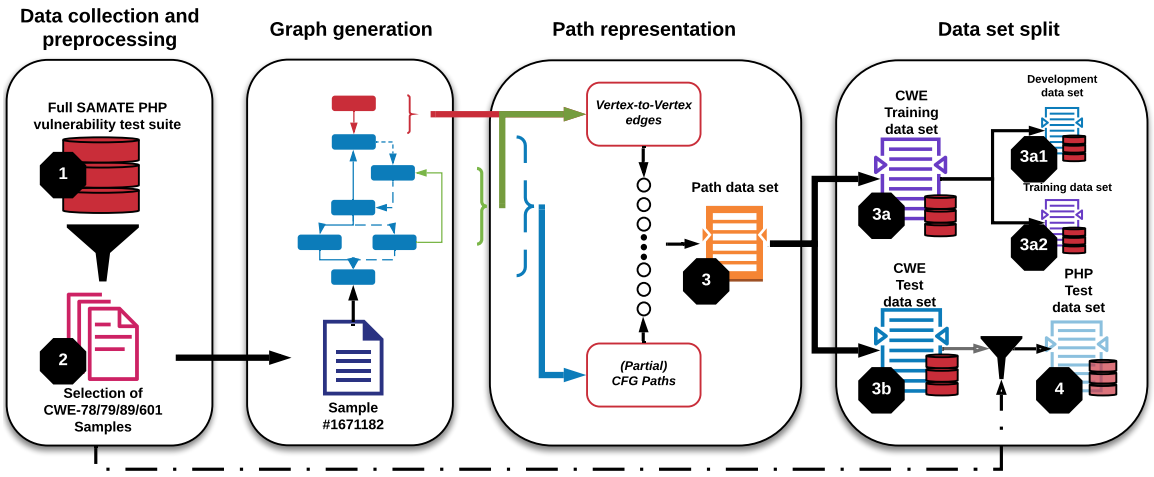


Figure 3.4: We select the samples pertaining to four classes of (injection) vulnerabilities (2) from the SAMATE PHP test suite (1). After generating graphs and extracting paths from these PHP files into path data sets (3), these four data sets are subsequently split in development (3a1), training (3a2), and test data sets (3b). For benchmarking purposes we also create a PHP test data set (4) populated with the PHP samples matching those in our path-based test data set.

The test data is kept separate from the training and development data. The training and development data set are chosen at random from the CWE training data set (3a) per iteration. Care is taken that the distributions of *safe* and *unsafe*-labeled samples in the development (3a1), training (3a2) and test data set (3b) are kept roughly equal to the distributions of the underlying data set (3). After our classifier development we will benchmark our models. For benchmarking we use the same PHP samples from the SAMATE PHP test suite as the ones used to generate our path-based data set, illustrated by data set 4 in figure 3.4. This guarantees a honest comparison of performance between our method and the performance of the tools used.
Using split training and test data sets is important as we want to evaluate the performance of the final model using samples in a data set which has not been used to train the model. Splitting guarantees that the model does not learn to recognise specific samples.

3.2.2. VULNERABILITY CLASSIFICATION USING DEEP LEARNING CLASSIFIERS

In order to predict vulnerabilities using the features generated in the previous step, we answer the following research question: *How can we develop a classifier, able to predict vulnerabilities based on features generated from code property graphs?*
We strive to find the best performing configuration for a densely connected feed-forward neural network per vulnerability category. To this end we train different classifiers for a number of vulnerability categories using the training data set shown in figure 3.4. This allows us to leverage the best possible performance per category. At a later moment, different models can be used together to predict multiple vulnerabilities based on the same graph based data.
In order to find the configuration of hyperparameters which yields a classifier which outperforms others we do a *grid search* to discern patterns in hyperparameters, top performing hyperparameters are subsequently tuned in a manual tuning phase to yield the final

classifier.

GRID SEARCH

To get an intuition for which hyperparameters might yield a promising classifier, we do a grid search. In this grid search we generate a 'grid' of configurations for every combination of predefined ranges for each hyperparameter (see table 3.4). We then train and test classifiers from this grid and score the best version according to an appropriate metric. This metric depends greatly on the available data set for the vulnerability. Any model unable to attain scores higher than a random classifier (i.e. F_1-scores > 0.5 and ROC-AUC > 0.5) will be discarded at this point. For the grid search we adhere to the following two principals. Firstly, we observe the whole grid, looking for trends as opposed to a single optimal solution. Secondly, we start coarse grained and zoom into finer grained grids as we move along. The most granular searches are done by manually manipulating the hyperparameters individually.

Three hyperparameters in the grid search are limited to a single default value as table 3.4 shows, the activation, optimization and loss functions. As explained in section 2.4.1 the ReLu is considered the 'golden standard' when it comes to activation functions for the hidden layers. We feel it is safe to limit the grid search to only using ReLu activators. As far as the optimization function is concerned, we set the default value to Adaptive Moment Estimation (Adam). Adam is a popular optimization function which has several merits, which makes it preferable over stochastic gradient decent. [34] Our main consideration in using Adam is that we will not need to pay as much attention to choosing a correct learning rate at the start, as Adam will be able to correct the learning rate based on the training process. Dropout is implemented as a separate hidden layer, which in our case is always placed as the first hidden layer, disabling a certain percentage of the inputs.

Table 3.4: Hyperparameter ranges for the grid searches

Parameter	**Range**
Hidden layers	[2, 3, 5, 7, 9, 11, 13]
Units per layer (shape)[3]	[linear, curve]
Dropout	[10%, 15%, 20%]
Activation function	ReLu
Epochs	[5, 10, 20, 50, 150, 200]
Batch size	[32, 64, 128, 256, 512]
Learning rate	[0.0005, 0.001, 0.0015, 0.002]
Optimization function	Adam
Loss function	Binary cross-entropy

The full grid consists of 5040 separate configurations. These configurations are stored in a file so we might reuse the configurations at a later point. Training and evaluating 5040 different configurations would take an exorbitant amount of time. We reduce this time by (randomly) sampling 200 configurations before evaluating which hyperparameter

[3] We aggregate the size of each hidden layer as the *shape* of the model. A linear model has a linear decline from the size of the input layer to the size of the ouput layer, which is 1. The layers size is computed as follows: $w_i = W - \lfloor W/I \rfloor * i$, where w_i is the size of layer i, I is the total number of hidden layers and W is the size of the input layer. The curved shape has a greater decline in the first layers but levels off near the output layer. We compute these sizes as follows: $w_i = \lceil (i+1) * W * e^{-i} \rceil$.

values consistently undderperform. Figure 3.5 shows this strategy. After training 200 of the possible configurations, we conclude that models with 5-10 hidden layers with a curved shape outperform other models. This allows us to no longer consider linear models, or models with 2, 3, 11 or 13 hidden layers for the rest of the grid search.

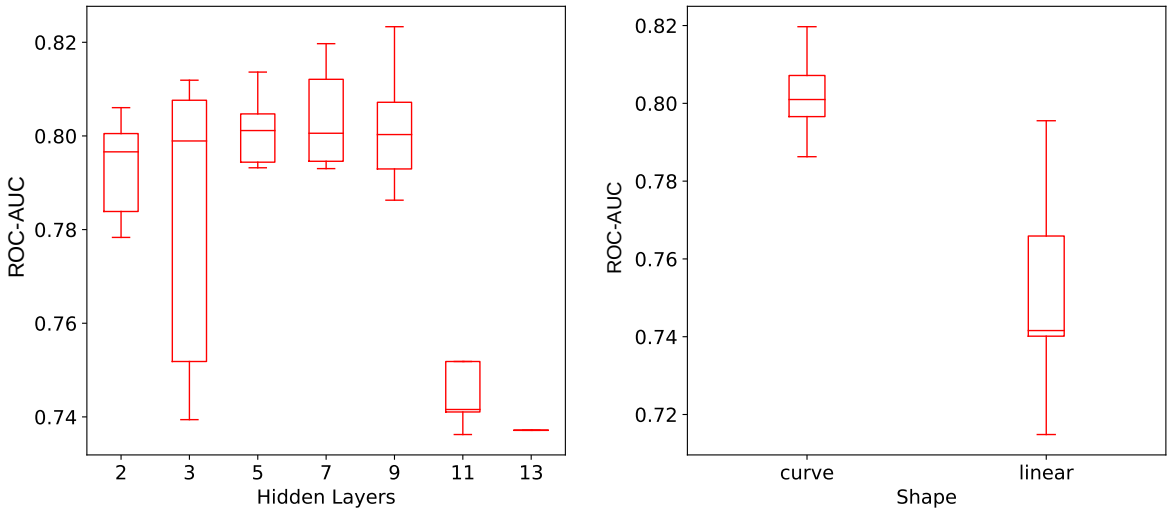


Figure 3.5: Trends identified early in the grid search, allow for reliable reduction of the search space.

As the grid search progresses we monitor the performance of the models to develop an intuition for which values in table 3.4 will and will not likely yield promising values. If analysis shows that certain hyperparameter values will most likely not yield high performance, these will be excluded from the ongoing grid search. This method greatly reduces the number of configurations which need to be tested.
For the grid search we rely solely on the training and development data set. We train the model using the training data set, after each epoch we track the performance of the model-in-training using a small portion of the training data (15%) as a validation data set and we use the development data set to measure the performance of the model after training is complete.

MANUAL FINE TUNING

The grid search presents us with an intuition for which parameters, from the selection in table 3.4, yield the best models. As the possible values of the parameters in the grid leave some gaps, we need to consider values not included in the aforementioned table. We do this by taking the optimal performing model according to the grid search and, subsequently tuning each hyperparameter separately following the trends found in the grid search. The resulting configuration will be used to train our optimised (final) model for this vulnerability category. Because results can vary due to a certain degree of entropy in training we run several experiments before drawing a conclusion on whether to focus on or discard a certain hyperparameter value.
In the manual fine tuning we will make use of the same training and development data sets as we have used in the grid search. This ensures that we can objectively compare the performance of our tuned models with the performance of the grid search models.

Having developed models for each category of vulnerabilities which yield good results, we test these models. We combine and shuffle the training and development data sets used in the grid search and manual tuning steps to create a new training data set. We measure the performance of our final model by using the unused test data set.

3.2.3. EVALUATING THE PERFORMANCE OF OUR CLASSIFIERS

We benchmark our classifiers in order to answer the following research question: *How does the proposed method perform compared to other automated approaches to vulnerability detection (i.e. open-source tools and comparable research)?*

We compare the results of our classifiers to results obtained from several open source vulnerability scanners. Firstly we use the output generated by Pixy[43][4] to compare the results of our classifiers for SQL Injection and Cross-Site Scripting. Pixy, as many vulnerability scanners do, uses data flow analysis to track possibly tainted data through the control flow of a program. If potentially tainted input reaches a so called 'sink' (i.e. a point where the data is used to perform for instance a query or call a system function), without being sanitised, the source code is deemed vulnerable by the scanner. As Pixy is unable to detect open redirect or OS command injection type vulnerabilities, we are not able to compare these specific results.
Secondly we evaluate results generated by RIPS(v.0.55)[5], another static (data flow) analyzer. RIPS uses a similar method to Pixy to detect different types of vulnerabilities including Cross-Site Scripting, SQL Injection, HTTP Response Splitting, File Disclosure, PHP Object Injection and Code Execution.
In generating the results for RIPS and Pixy, we use the same test data set as we have used in testing our classifiers. This allows us to directly compare the results of the tools to our classifiers.
We also compare the performance of our classifiers to the performance of WIRECAML[16][6], WAP[7], and Yasca[8]), based on the results described by Kronjee in his thesis "Discovering vulnerabilities using data-flow analysis and machine learning." We were unable to redo the tests with our test data set due to mismatches in dependencies. We do however feel safe to rely on the results described by Kronjee as they too, are based on the SAMATE PHP test suite. Note that the results described by Kronjee are limited to SQL Injection and Cross site scripting only.

3.3. RESEARCH CONTRIBUTION

In this research, we explore whether graph representations of source code can be used in a deep learning approach to detect vulnerable source code. We build on work by Kronjee and Yamaguchi, Golde, Arp, *et al.* by using characteristics derived from code property graphs as indicator for vulnerable elements in source code. Kronjee and Yamaguchi, Golde, Arp, *et al.* base their detection of vulnerable code on specific sink to source traces, matching a predefined pattern.

[4] https://github.com/oliverklee/pixy
[5] http://rips-scanner.sourceforge.net/
[6] https://github.com/jorkro/wirecaml
[7] http://awap.sourceforge.net/
[8] https://github.com/scovetta/yasca

Our method significantly differs from these approaches in that we depend on a deep learning classifier to indicate which syntactical, structural and semantic aspects of the code property graph represent vulnerabilities. With little to no need to manually identify specific elements in a given programming language representing a given vulnerability our method can be applied easily across a broader spectrum of vulnerabilities and programming languages. With the method used to construct our PHP classifiers serving as an example, we enable straightforward development of similar classifiers for other types of vulnerabilities and other programming languages.
Our primary research contribution is a well founded answer to our research question *"How can we effectively predict the presence of injection vulnerabilities using features learned from graph representations of source code?"* in the form of this thesis, and the results on which the answer is founded. Secondarily we will make tooling and models available to enable others to build on our research.
Our first deliverable is the tooling to convert PHP files to code property graphs in the form of a `CodePropertyGraph` python module. From the graphs generated using this module, we subsequently extract edges and paths into a data set for developing our classifiers. The aforementioned module includes methods to extract these elements from the graph. We develop a classifier for each category of vulnerability in 3.1. We will make the both the tooling used to develop the classifiers and the models themselves available.
Used in unison, this tooling enables one to retrace the steps in this research for other (PHP) data sets, or to use the classifiers to predict vulnerabilities in (PHP) projects. Furthermore the tooling in conjunction with this thesis might serve as a basis to develop new classifiers for other vulnerability categories or other programming languages.
Tools will be made available as Jupyter notebook[9] under the MIT-licence. Classifiers will be made available as *hierarchical data format* file (HDF5), also under the MIT-licence.

[9]Jupyter notebook is an open-source browser-based, python IDE, available from `https://jupyter.org/`

4

CLASSIFIER DEVELOPMENT

As described in chapter 3, we develop four classifiers in total; one for each category of vulnerabilities in our selection (OS command injection, Cross site scripting, SQL-injection, and Open redirect). In developing these classifiers we follow the same four steps.

1. We perform a partial grid search by randomly selecting 200 of the 5040 possible configurations to train classifiers;
2. The scores of these classifiers are evaluated in an attempt to discern patterns with respect to their hyperparameter values (e.g. *"do models with 7 or more hidden layers generally outperform models with fewer hidden layers?"*). Based on these conclusions we remove configurations with underperforming hyperparameters from the grid;
3. We train and evaluate the classifiers in the remaining grid;
4. From the classifiers we select the top-performing configuration to retrain and subsequently evaluate its final performance using our test data set.

In developing these classifiers we name them after chemical elements for easy reference. Sections 4.1.1 to 4.1.4 feature both the optimised configuration for the individual classifiers (Cerium, Nitrogen, Lithium, and Arsenic), as well as their respective test scores. Section 4.2 concludes the chapter with some general observations.

4.1. CLASSIFIERS PER VULNERABILITY

As noted in section 3.2.1 each category of vulnerability will require a different classifier. As a consequence we have split the data set based on the vulnerability categories. Table 3.1 shows that not every data set is equally balanced between *safe* and *unsafe* samples. To be able to successfully train the classifiers with data sets with a limited portion of unsafe samples (i.e. OS command injection and SQL-injection (SQLi)), we apply K-fold cross validation during classifier development to ensure that every observation from the original data set has the chance of appearing in training and test set. We choose to monitor the (weighted average) F_1-scores during classifier development. We will use a classification threshold of 0.5 for all the classifiers. For each classifier we include a classification report, indicating the performance of the classifier based on test data. This classification report includes a (macro) average for precision, recall and F_1-scores and a weighted average where

the number of occurrences per class are incorporated in the average. We also include a plot for the primary metric (PR) used in developing the classifier, indicating the trade-off between precision and recall for different thresholds.

4.1.1. CLASSIFIER FOR OS COMMAND INJECTION (CWE-78)

We train the *Cerium* configuration against the full training set, and use the test data set to measure performance.

***Cerium* configuration**	
Hidden layers	5
Learning rate	0.001
Dropout	20%
Epochs	5
Batch Size	128
Shape	curve
Optimizer	*Adam*
Activation function	*ReLu*
Loss function	*Binary cross-entropy*

Table 4.1: Optimised configuration for *Cerium*. Note that the optimizer function, loss function and activation function for the hidden layers were preset to Adam, binary cross-entropy, and ReLu respectively.

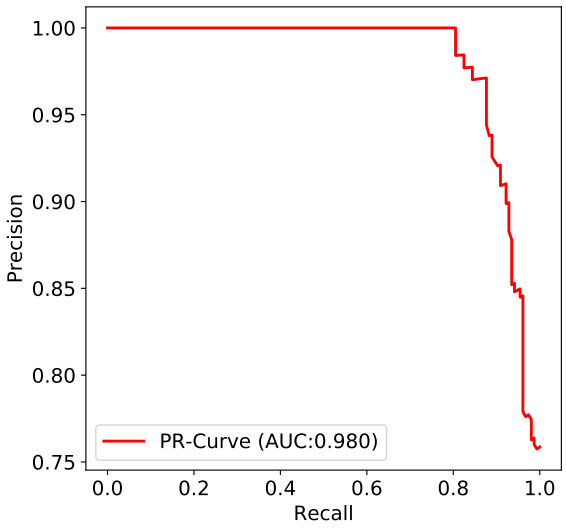


Figure 4.1: PR-curve for *Cerium*

Cerium PERFORMANCE

Table 4.2: Classification report and confusion matrix for *Cerium*

	Classification report			**Confusion matrix**		**Support**
	Precision	**Recall**	**F_1-score**	**Predicted unsafe**	**Predicted safe**	
Unsafe	0.714	0.714	0.714	**35**	14	49
Safe	0.909	0.909	0.909	14	**140**	154
Accuracy	0.862			49	154	**203**
Macro avg	0.812	0.812	0.812			
Weigthed avg	0.862	0.862	0.862			
ROC-AUC	0.93					
PR-AUC	0.980					

Considering the results for Cerium, it seems that the unbalanced data set, despite our best efforts to counter its effects, does have an impact on the stability of the classifier indicated by the significant difference between F_1-scores for *safe* and *unsafe* classes.

4.1.2. CLASSIFIER FOR CROSS SITE SCRIPTING (CWE-79)

The configuration resulting from the grid search and manual tuning, which we dub *Nitrogen* is described in table 4.3.

***Nitrogen* configuration**	
Hidden layers	7
Learning rate	0.00325
Dropout	30%
Epochs	4
Batch Size	70
Shape	curve
Optimizer	*Adam*
Activation function	*ReLu*
Loss function	*Binary cross-entropy*

Table 4.3: Optimised configuration for *Nitrogen*. Note that the optimizer function, loss function and activation function for the hidden layers were preset to Adam, binary cross-entropy, and ReLu respectively.

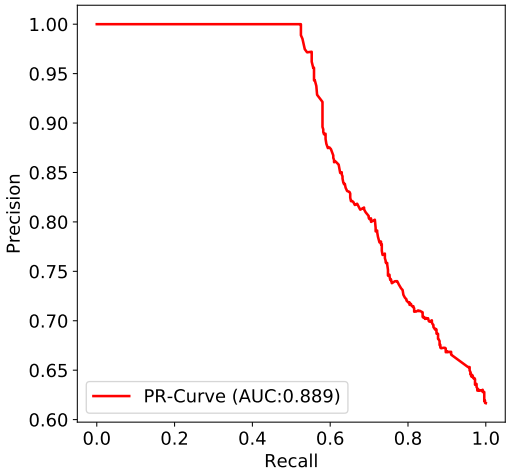


Figure 4.2: PR-curve for *Nitrogen*.

Nitrogen PERFORMANCE

Table 4.4 lists the classification report and confusion matrix for our *Nitrogen* model.

Nitrogen PERFORMANCE

Table 4.4: Classification report and confusion matrix for *Nitrogen*

Classification report				**Confusion matrix**		**Support**
	Precision	**Recall**	**F_1-score**	**Predicted unsafe**	**Predicted safe**	
Unsafe	0.638	0.828	0.720	**317**	66	383
Safe	0.831	0.644	0.725	180	**325**	505
Accuracy	0.723			497	391	**888**
Macro avg	0.735	0.736	0.723			
Weigthed avg	0.748	0.723	0.723			
ROC-AUC	0.830					
PR-AUC	0.889					

In contrast to Cerium, Nitrogen does not suffer from a significant difference between the ability to accurately and precisely classify *safe* or *unsafe* samples, supporting our hypothesis that the root cause is the unbalanced training set.

4.1.3. CLASSIFIER FOR SQL-INJECTION (CWE-89)

***Lithium* configuration**	
Hidden layers	9
Learning rate	0.002
Dropout	15%
Epochs	50
Batch Size	32
Shape	curve
Optimizer	*Adam*
Activation function	*ReLu*
Loss function	*Binary cross-entropy*

Table 4.5: Optimised configuration for *Lithium*. Note that the optimizer function, loss function and activation function for the hidden layers were preset to Adam, binary cross-entropy, and ReLu respectively.

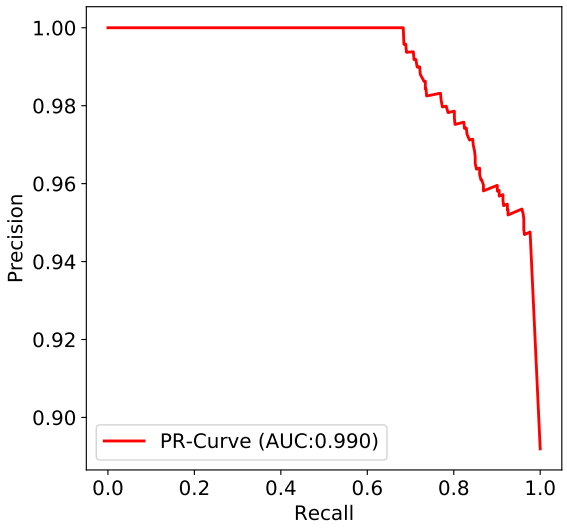


Figure 4.3: PR-curve for *Lithium*

Lithium PERFORMANCE

Table 4.6: Classification report and confusion matrix for *Lithium*

Classification report				**Confusion matrix**		**Support**
	Precision	**Recall**	**F_1-score**	**Predicted unsafe**	**Predicted safe**	
Unsafe	0.742	0.554	0.634	**46**	37	83
Safe	0.948	0.977	0.962	16	**669**	685
Accuracy	0.931			62	706	**768**
Macro avg	0.845	0.765	0.798			
Weigthed avg	0.925	0.931	0.927			
ROC-AUC	0.919					
PR-AUC	0.990					

The configuration for Lithium differs significantly from its counterparts. It is trained for many more epochs, has a much smaller batch size, and features more hidden layers. Furthermore the difference in scores (based on $F_1(unsafe)$) is quite telling. We believe the primary reason to be that the data set for SQLi has many more features per sample and line of code than the other data sets (see fig. 3.3). It would appear that a manifestation of a SQL injection vulnerability in source code requires more code, yielding more features, requiring a more intricate neural network to detect successfully.

4.1.4. CLASSIFIER FOR OPEN REDIRECT (CWE-601)

Arsenic configuration	
Hidden layers	7
Learning rate	0.002
Dropout	20%
Epochs	5
Batch Size	64
Shape	curve
Optimizer	*Adam*
Activation function	*ReLu*
Loss function	*Binary cross-entropy*

Table 4.7: Optimised configuration for *Arsenic*. Note that the optimizer function, loss function and activation function for the hidden layers were preset to Adam, binary cross-entropy, and ReLu respectively.

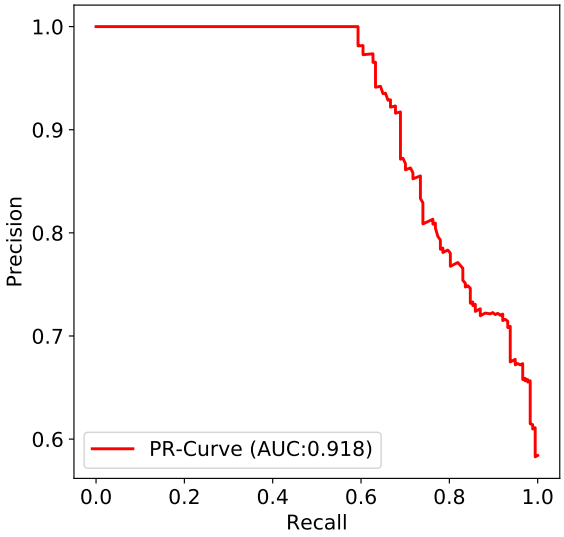


Figure 4.4: PR-curve for *Arsenic*

Arsenic PERFORMANCE

Table 4.8: Classification report and confusion matrix for *Arsenic*

	Classification report			Confusion matrix		Support
	Precision	**Recall**	**F_1-score**	**Predicted unsafe**	**Predicted safe**	
Unsafe	0.784	0.939	0.855	**200**	13	213
Safe	0.904	0.689	0.782	55	**122**	177
Accuracy	0.826			255	135	**390**
Macro avg	0.844	0.814	0.818			
Weigthed avg	0.838	0.826	0.822			
ROC-AUC	0.919					
PR-AUC	0.918					

Even though open redirect is not formally classified as an injection vulnerability, we were able to train a classifier which performs well using the same methodology as we have used with Lithium, Cerium and Nitrogen. This indicates that our method for graph representations of source code and for classifier development can be used outside the formal scope of injection vulnerabilities.

4.2. CONCLUSION

Table 4.9 shows a summary of our classifiers, their configurations, and their performance.

Table 4.9: Summarising the different vulnerability classifiers.

Name	CWE	HL	LR	Drop-out	Ep.	Batch Size	Weighted avg. F_1	F_1 Unsafe	F_1 Safe
Cerium	78	5	0.001	20%	5	128	0.862	0.714	0.909
Nitrogen	79	7	0.00325	30%	4	70	0.723	0.720	0.725
Lithium	89	9	0.002	15%	50	32	0.927	0.634	0.962
Arsenic	601	7	0.002	20%	5	64	0.822	0.855	0.782

HL: Hidden Layers, LR: Learning rate, Ep.: Epochs.

Note that all of the configurations are based on a curved 'shape', Adam optimizers, ReLu activation functions in hidden layers, and a binary cross-entropy loss function.

When inspecting the scores in general, our method appears to be sound, only yielding classifiers which score quite high looking at their respective F_1 and AUC-scores. This indicates that the Code Property Graphs and the features we generate from them are sufficiently expressive so that they might be leveraged to train classifiers and be used to predict vulnerabilities. We will explore how our classifiers compare to other methods of vulnerability prediction in the next chapter.

4.2.1. LIMITS TO THE GRID SEARCH

We are aware that, by limiting our grid to the values in table 3.4, we might miss some optimal configurations with hyperparameter values which are not incorporated in the grid. This applies to two cases: on the one side when trends indicate that hyperparameter values fall outside the grid (e.g. more than 13 hidden layers). The other case is concerned with the intermediary hyperparameter values (e.g. 6 hidden layers).

Regarding the former case, all of the trends we were able to discern in regard to the hyperparameter values, manifested themselves contained within the values dictated by the original grid. None of the trends gave cause to expand the grid beyond the values described in table 3.4.

Regarding the latter case, we set out to prevent missing these intermediate values by applying a manual tuning step following the grid search as described in section 3.2.2, after which we explore other hyperparameter values close to the configuration which yields the optimal after the grid search.

Table 4.10: Manual tuning: Nitrogen versus Tellurium

Name	CWE	HL	LR	Drop-out	Ep.	Batch Size	Weighted avg. F_1	F_1 Unsafe	F_1 Safe
Tellurium	79	7	0.001	15%	5	64	0.722	0.721	0.723
Nitrogen	79	7	0.00325	30%	4	70	0.723	0.720	0.725

HL: Hidden Layers, LR: Learning rate, Ep.: Epochs.

Note that all of the configurations are based on a curved 'shape', Adam optimizers, ReLu activation functions in hidden layers, and a binary cross-entropy loss function.

However when applying this second step when developing a classifier for XSS, we conclude that the manual tuning step does not add significant improvements in performance. The grid search for XSS initially led us to a model dubbed *Tellurium* before finally settling on the (*Nitrogen*) configuration after manual tuning.
When comparing weighted average F_1-scores for *Tellurium* and *Nitrogen*, we noticed that the difference is negligible with Tellurium scoring 0.722 versus the tuned version, Nitrogen, scoring 0.723 (see table 4.10).
Based on this finding we feel safe in omitting the manual tuning step in developing the other classifiers and choosing the top performing classifier based solely on the results of the grid search.

5

CLASSIFIER COMPARISON

In this chapter we attempt to answer the question *How does the proposed method perform compared to other automated approaches to vulnerability detection (i.e. open-source tools and comparable research)?* To do so we compare the performance of our four classifiers to the performance of other solutions sharing the same goal of predicting the presence of a specific type of vulnerability in source code.

5.1. TOOLS

We compare the performance of our classifiers to the results on identical or similar data sets by other tools.
We have selected three tools for this comparison, RIPS, Pixy and WIRECAML. In addition to the aforementioned tools, we had intended to compare the performance of our classifiers agains the performance of Yasca[1] and WAP[2]. Both tools are open source static analyzers using data flow analysis. Yasca allows for several plugins to be added to a core program, which allows it to provide better judgment on vulnerability of PHP source code. We were however unable to run either one of these tools. The low scores described by Kronjee, has led us to omit both tools from the results as they under perform strongly to WIRECAML.
In the next subsections we discuss each of the tools used and we will indicate which data sets are used. Note that the origin of each individual data set is described in figure 3.4. We provide a summary of which tool–data set combinations were used in table 5.1.

5.1.1. RIPS

RIPS has been a popular open source tool for static PHP vulnerability analysis for several years. RIPS works by tracing user input to potentially vulnerable functions, referred to as sinks, in the source code. As explained in section 2.1, tainted data introduced to a sink could lead to a system being compromised.
The publicly available version 0.55 of RIPS[3] offers a web interface from where users can start the analyses. Although a newer (commercial) version of RIPS is available[4] we were unable to obtain a copy for evaluation purposes.

[1] https://github.com/scovetta/yasca
[2] http://awap.sourceforge.net/
[3] http://rips-scanner.sourceforge.net/
[4] https://www.ripstech.com/

RIPS is built to analyse the PHP source code files. For our evaluation of the performance of RIPS we have selected those source code files from which we have built our own data sets. Results can therefor be directly compared to our own results.

5.1.2. PIXY

Pixy[5], as many vulnerability scanners do, uses data flow analysis to track possibly tainted data through the control flow of a program. [43] If potentially tainted input reaches a sink, without being sanitised, the source code is deemed vulnerable by the scanner. As Pixy is unable to detect open redirect or os command injection type vulnerabilities, we are not able to compare these specific results.

Table 5.1: Origin of evaluation data per tool used.

Tool	**SQL Injection** CWE-89	**XSS** CWE-79	**Command Injection** CWE-78	**Open Redirect** CWE-601	**Data set**
RIPS	✓	✓	✓	✓	PHP test data set†
Pixy	✓	✓			PHP test data set†
WIRECAML*	✓	✓			SAMATE PHP test suite

† *This data set is identical to the data set used in developing and evaluating our classifiers.*

**These results were obtained from Kronjee. [16]*

5.1.3. WIRECAML

We have based this research in part on Kronjee's Masters Thesis *"Discovering vulnerabilities using data-flow analysis and machine learning."*. [16] Kronjee also makes use of the SAMATE PHP test suite to evaluate the performance of his classifier *WIRECAML*. The evaluation of WIRECAML is based on the SAMATE PHP test suite samples pertaining to SQLi and XSS vulnerabilities. The data set used in developing our own classifiers is based on a subset of these samples. As such we are able to copy his results to allow us to compare our classifiers to WIRECAML. [16, p. 42]
WIRECAML is limited to detecting SQLi and XSS. OS command injection (CWE-78) and open redirect (CWE-601) have not been implemented and will therefor not be compared.

5.2. RESULTS

We aggregate the results of each tool for each type of vulnerability. For each class of vulnerability we provide the weighted average precision and recall and the individual F_1-scores for *safe* and *unsafe* classes.

5.2.1. OS COMMAND INJECTION (CWE-78)

Table 5.2 clearly shows that Cerium outperforms the other tools on every metric. Cerium shows to be able to accurately (86%) predict the presence of unsafe constructs in the presented code. RIPS shows to be incapable of accurately detecting Command injection vulnerabilities in our test data set.

[5] https://github.com/oliverklee/pixy

Table 5.2: OS Command injection score comparison

Tool	Precision	Recall	F_1-score unsafe	F_1-score safe
Cerium	**0.86**	**0.86**	**0.71**	**0.91**
RIPS	0.18	0.04	0.07	0.86

5.2.2. CROSS-SITE SCRIPTING (CWE-79)

With this category, our classifier Nitrogen performs adequately, attaining a F_1-scores of 0.72 and 0.73 for unsafe and safe classes respectively, which indicates it outperforms other methods. RIPS and Pixy fail to yield a significant performance, with RIPS classifying every single sample as *unsafe*. and Pixy only slightly outperforming the random classifier. WIRECAML shows stable performance well above the random classifier. Note that Nitrogen and WIRECAML follow a similar pattern, with a slightly higher precision and a surprisingly uniform Recall and F_1-scores. This pattern is what we would expect to see with data sets with a high level of similarity. This legitimizes our choice to base our comparison on the scoring supplied by Kronjee even though the testing data sets are not 100% identical. [16]

Table 5.3: XSS score comparison

Tool	Precision	Recall	F_1-score unsafe	F_1-score safe
Nitrogen	0.75	0.72	**0.72**	**0.73**
RIPS*	0.43	**1.00**	0.60	0.00
Pixy	0.51	0.48	0.50	0.63
WIRECAML	**0.79**	0.71	0.71	0.70

**RIPS has not found any safe samples, which results in a recall of 1 and F_1-score for safe samples of 0.*

5.2.3. SQL-INJECTION (CWE-89)

Table 5.4 shows that WIRECAML scores highest across the board for SQLi, with Lithium only slightly trailing in every category. Here, similar to the patterns seen with WIRECAML and Lithium for XSS, there are many similarities between the two, confirming again that our choice for copying the scores directly from Kronjee is sound. As far as RIPS and Pixy go, the conclusions for XSS also apply here. Note that in this case RIPS is unable to detect any *unsafe* samples.

Table 5.4: SQLi score comparison

Tool	Precision	Recall	F_1-score unsafe	F_1-score safe
Lithium	0.93	0.93	0.63	0.96
RIPS*	0.50	0.00	0.00	0.94
Pixy	0.22	0.58	0.32	0.70
WIRECAML	**0.94**	**0.94**	**0.66**	**0.97**

**RIPS has not found any unsafe samples, which results in a recall of 0 and F_1-score for unsafe samples of 0.*

5.2.4. OPEN REDIRECT (CWE-601)

The performance of Arsenic shows that our approach can be used for vulnerabilities which are not formally classified as injection vulnerabilities. Arsenic outclasses the performance RIPS entirely as table 5.5 shows.

Table 5.5: Open redirect score comparison

Tool	Precision	Recall	F_1-score unsafe	F_1-score safe
Arsenic	**0.84**	**0.83**	**0.86**	**0.78**
RIPS	0.62	0.23	0.34	0.58

5.3. CONCLUSION

The results of our comparisons show that our series of classifiers generally perform well on the different data sets. When inspecting the ability to accurately and precisely predict the presence of *unsafe* source code, which we see as the classifiers' primary objective, we find that, our models outperform or closely match other tools.

We do find that the ability to successfully predict the presence of vulnerabilities, represented by the F_1-score for the *unsafe* class decreases when the classifier is trained with fewer unsafe training samples. This is represented by the difference between the F_1-scores for the *safe* and *unsafe* classes for OS command injection and SQLi.

In comparing the results of our classifiers with WIRECAML, we chose to obtain the scores for WIRECAML from Kronjee. [16, p. 42] The scoring for WIRECAML might not be based on a 100% identical data set as we lose some samples from the SAMATE test suite in preprocessing. Scores for XSS and SQLi for both our classifiers and WIRECAML however show very similar patterns and score fairly close to each other, which we conclude to be a confirmation that both data sets are comparable and that our choice to copy these scores is sound.

6

DISCUSSION

Our final chapter discusses the outcome of our research, per research question before answering our central question: *How can we develop a classifier, able to predict vulnerabilities based on features generated from code property graphs?* We will also list our deliverables which we will make available. In the final sections we will discuss limitations, related and future work.

6.1. RESEARCH OUTCOME

Our research has led us to develop a variation of the code property graph as defined by Yamaguchi, Lottmann, and Rieck. Using our tooling we have converted samples from the SAMATE PHP vulnerability data set to these code property graphs. Paths extracted from these graphs were subsequently used to train, optimize and evaluate deep learning classifiers for each category of vulnerabilities in table 3.1. The classifiers were subsequently compared to other publicly available solutions (using the same data set where possible).
In our research we have strived to answer the following research question:"How can we effectively predict the presence of injection vulnerabilities using features learned from graph representations of source code?" From this question we derive the following sub-questions:

R.Q.1 *How can we represent Code Property Graphs so that they can be successfully exploited in a deep-learning approach?*

R.Q.2 *How can we develop a classifier, able to predict vulnerabilities based on features generated from code property graphs?*

R.Q.3 *How does the proposed method perform compared to other automated approaches to vulnerability detection (i.e. open-source tools and comparable research)?*

In the following sections we will describe the outcome for each of these questions separately.

6.1.1. R.Q. 1 REPRESENTING CODE PROPERTY GRAPHS

How can we represent Code Property Graphs so that they can be successfully exploited in a deep-learning approach?
Where Yamaguchi, Lottmann, and Rieck and Kronjee predefine features based on traces

within the graph, we propose a novel approach. We propose to extract edges and paths of a certain (parameterized) length between CFG nodes from these graphs following every possible control flow to allow a neural network to discern useful features from. To do so we present a new definition of the Code Property Graph, based on Yamaguchi, Lottmann, and Rieck. We simplify the CPG by defining a pruned version of the original.
As mentioned in section 3.1, the expressiveness of the extracted paths, and as a result our features, is generally a tradeoff between specificity and the models ability to generalize. We opt, for instance, for path lengths of 3 to 9. Longer path lengths will inadvertently make the model recognise more specific paths but will not make it generalize better. This tradeoff is also present in pruning the initial graph, we opt to prune all but a few of the AST nodes. More detailed pruning might result in models which perform better on a specific data set, but will as a result perform poorer on previously unseen samples. This tradeoff is linked in some respect to the availability of data; if more (diverse) data is available one might produce more specific features without loss of ability to generalise these results to other samples.
Our tooling to convert PHP-source code to code property graphs, and from there to data sets with paths, offers flexibility by parameterising the minimal and maximal path lengths and the possibility to adjust the pruning of the graph when desired. We believe that we have proposed a workable method to successfully extract meaningful path-based data points from code property graphs, on which deep learning algorithms might base vulnerability classification.

6.1.2. R.Q. 2 CLASSIFIER DEVELOPMENT

How can we develop a classifier, able to predict vulnerabilities based on features generated from code property graphs?

After having generated path-based data sets based on code property graphs we set out to develop four different classifiers for each of the vulnerabilities listed in table 3.1.
We have used a grid search to get an intuition as to which configurations (i.e. combinations of hyperparameters) will yield deep learning feed-forward classifiers with good performance, measured by several metrics. We randomly sample 200 of the possible configurations and reduce the queue of the unprocessed configurations based on these preliminary results. The top performing configuration of the full search is subsequently described in detail and used for evaluation.
After our initial intent to manually tune the models, we find, with developing our *Nitrogen* classifier that this (rather time consuming) step only yields marginally higher performance. We conclude that a grid search as we have described above yields a sufficiently representative classifier as is.
We further note that balanced data sets seem to yield more stable classifiers. Despite our efforts to counter the effects of a skewed data set, the scores of Cerium and Lithium show a fairly large difference between F_1-scores for *safe* and *unsafe* samples.
The results described in section 4.2 show that the classifier for SQLi (Lithium) is based on a significantly different configuration to achieve optimum performance, this indicates that different types of vulnerabilities require different configurations. Looking at the difference in the number of features per line of code (figure 3.3) indicates that the path length used to generate features from the code property graphs might also require differentiation per vulnerability.

Table 6.1: Summary of the performance of our vulnerability classifiers.

Name	CWE	ROC-AUC	PR-AUC	F_1 Unsafe	F_1 Safe	F_1 Weighted avg.
Cerium	78	0.930	0.980	0.714	0.909	0.862
Nitrogen	79	0.834	0.889	0.735	0.738	0.737
Lithium	89	0.919	0.990	0.634	0.962	0.927
Arsenic	601	0.919	0.918	0.855	0.782	0.822

We believe we have successfully managed to develop four effective vulnerability classifiers based on paths extracted from Code Property Graphs, which all show high levels of performance as seen in table 6.1.
We must note that one of the major drawbacks of the method for developing our classifiers in comparison to other (non-deep learning) methodologies is that training the classifier requires a large amount of training data. A traditional static-analysis method will be based on prior knowledge of a vulnerability which has to be translated into a detection method. Our method on the other hand will need to infer this knowledge from data, which takes a lot of sample data, computing power and time. On the other hand, our method does not rely specifically on the ability to be able to correctly translate the properties of a certain vulnerability to a detection rule. It is likely that our method used to develop these classifiers is also able to yield effective vulnerability classifiers for other (similar) types of vulnerabilities and furthermore for other programming languages, if a sufficient amount of training data is available.

6.1.3. R.Q. 3 CLASSIFIER COMPARISON

How does the proposed method perform compared to other automated approaches to vulnerability detection (i.e. open-source tools and comparable research)?

We have compared the performance of our classifiers to the performance of the open source vulnerability scanners, RIPS, Pixy and WIRECAML. We have used the identical PHP data sets to compare the performance of our classifiers to those of RIPS, and Pixy (see figure 3.4). For the comparison with WIRECAML we base our conclusions on the results described by Kronjee. [16]
The results of our comparisons show that our series of classifiers generally perform well on the different data sets. When inspecting the ability to accurately and precisely predict the presence of *unsafe* source code, which we see as the classifiers' primary objective, we find that, our models outperform or closely match other tools. Furthermore our classifiers score quite high on other metrics.
We believe we have been able to show that classifiers, developed using our method, are able to closely match or outperform contemporary tools designed for vulnerability detection. We also note that the performance of our classifiers greatly depends on the available data for training. It must be noted that our classifiers, in contrast to RIPS, Pixy and WIRECAML, have not been tested with data from outside the SAMATE test suite. We will elaborate more on this limitation in section 6.3.2.

6.1.4. RESEARCH QUESTION AND ANSWER

How can we effectively predict the presence of injection vulnerabilities using features learned from graph representations of source code?

We have shown that by using our novel method we are able to extract meaningful data from PHP-based code property graphs. We have shown to be able to train and evaluate classifiers based on this data and have shown that our classifiers outperform or closely match other tools specifically designed for vulnerability detection. In sum, we believe that, based on the answers on the sub-questions in sections 6.1.1, 6.1.2, and 6.1.3 we have conclusively answered our research question by demonstrating the contribution our method can have to vulnerability research.

6.2. RESEARCH CONTRIBUTIONS

We believe that the most important research contribution our method offers in comparison to other described methods, is that our method allows us to develop effective classifiers for any related type of vulnerabilities using the exact same methodology of data (pre-)processing and classifier development *without* specific knowledge of the vulnerability at hand. Aside from a vulnerability-specific data set, the method requires little to no customisation to be used in developing classifiers for other types of vulnerabilities. We demonstrate this with our Arsenic classifier, which was developed using an identical method to Cerium, Lithium and Nitrogen, without any vulnerability-specific customisation of the dataset or to the method in which the classifier was developed. Other methods might require a security researcher to predefine potentially vulnerable functions for a specific type of vulnerablity, our method has no such dependency; our classifiers will infer the potential volatility of these functions from the training data.

Besides this methodology we plan to make the Python 3 `CodePropertyGraph`-module available which will convert PHP source code to our version of the (pruned) Code Property Graph. This module also allows the user to generate path-based data sets from these graphs. Annotations in the source code should enable researchers to develop similar versions of this module for different programming languages. We will also share Jupyter notebooks which implement the grid search used for our classifier development and evaluation, these can be used directly to develop new classifiers based on different data sets.

We will also make our pre-trained classifiers available as hierarchical data format file (HDF5). This will allow others to directly use the classifiers described in chapter 4.

Lastly we make this report, outlining our research, outcomes and decisions, available via the online library of the Open University.

6.3. LIMITATIONS

In our research we have occasionally run into a number of limitations, which we will describe in the sections below.

6.3.1. PROCESSING POWER

For our research we have used Google Colaboratory[1] (Colab) which provides free access to the Google Compute Backend via a web-interface based on the Jupyter Notebook. Colab

[1] https://colab.research.google.com/

gives us access to 25GB RAM, 68GB cloud storage, integrates Google Drive[2] (Drive) support and gives us the option to use hardware acceleration via GPU or TPU free of charge.
Use of hardware acceleration on Colab is, however restricted to twelve hours per day. Furthermore Colab requires constant monitoring to prevent the interface from timing-out. This limits how we execute the grid search and fine tuning. Would we have had more resources available, we might have opted to include more hyperparameter values in the grid search. This would have guaranteed that would find the optimal configuration by solely relying on the outcome of the (full) grid search.

6.3.2. DATA SET

Because of time restrictions we have opted to only use the SAMATE test suite, as described in chapter 4. The data set is computer generated, which might result in the source code containing elements, in either syntax or structure, specific to the method of generation. If our method is trained to recognise these specific artefacts, our model will not be able to generalize; it might not be able to accurately categorize samples which were not in the SAMATE test suite.
We believe that, because of the translation of the source code to the code property graphs, abstracts away from most of the artifacts which might be left over from the generation process largely mitigating these unwanted effects. Unwanted effects from the data set can however not be completely discounted.

6.4. RELATED WORK

As noted in chapter 2 vulnerability research using machine or deep learning approaches are gaining interest. We have often referenced the work of Yamaguchi, Lottmann, and Rieck and Kronjee as the basis for our research. [1][16] Both studies have relied on the use of Code Property graphs as a basis for their research and have obtained admirable results with them.
Using the expressive power of Code Property Graphs, Xiaomeng, Tao, Runpu, *et al.* have obtained good results, predicting vulnerabilities in C/C++ source code based Code Property Graphs by using a combination of classifiers built with recurrent neural networks (RNN). [40] Combining classifiers might be of added use to amend our method as well.
Other approaches, not focusing on the use of Code Property Graphs, but rather concentrating on the textual representations of source code also seem promising. The research by Li, Zou, Xu, *et al.*, "VulDeePecker: A deep learning-based system for vulnerability detection" breaks source code up in semantically related "gadgets" which are subsequently vectorised and used in a Binary Long Short-Term Memory (BLSTM) model to achieve commendable results. [35]
Recent research by Wang, Liu, and Tan proposes to leverage Deep Belief Network (DBN) to automatically learn semantic features from token vectors extracted from programs' Abstract Syntax Trees to predict defects in software. [44]
A similar approach is proposed by Russell, Kim, Hamilton, *et al.* in "Automated Vulnerability Detection in Source Code Using Deep Representation Learning", where they tokenize source code before applying a convolutional filter to the data set. [45] Their method enables the authors to detect and indicate where vulnerabilities in source code are situated,

[2] https://drive.google.com/

an added benefit compared to our own method.
Dam, Tran, Pham, *et al.* also applies tokenizing, followed by K-means clustering before applying a Long Short-Term Memory model (LSTM) to detect vulnerabilities. Their research is particularly interesting as they successfully apply their method to real world Android applications to validate their work. [46]

6.5. FUTURE WORK

Our work proves the feasibility of using deep learning classifiers to predict vulnerabilities in source code using graph representations. Our approach leaves a number of opportunities to expand on or vary on our method.
A future implementation of our method might consist of a given piece of source code being assessed by multiple (vulnerability specific) classifiers to reach a verdict on which vulnerabilities are present. A researcher can vary in the scope he tests – where one might start out with evaluating files and, based on the outcome for a given file, do a more granular (e.g. function level) assessment to narrow down where the vulnerability is present. Care must however be taken in preserving the origin of variables introduced by users as they are key in identifying possible flaws.
One obvious variation to our research is to apply our method to detect vulnerabilities in samples written in a different programming language to PHP. Most programming languages are suited to converting source code to AST, CFG and/or PDG, which opens the door for constructing a Code Property Graph after which our method for classifier development can be applied without modification. Furthermore, future work might explore the applicability of our approach to different vulnerabilities. Overflow vulnerabilities for instance are based on similar concepts to injection vulnerabilities and might be detectable using a classifier developed using our method.
As we have noted, we have limited the feature generation on path lengths to a maximum length of nine edges. We expect different vulnerability types to perform differently when varying this path length parameter. Future work could investigate this hypothesis and attempt to establish an optimal path length for every type of vulnerability.
An interesting variation would be to implement different deep-learning models in classifier development. Perhaps the performance of our densely connected feed-forward networks can be surpassed by other topologies such as Recurrent Neural Networks (RNN) or LSTM.
We strongly believe that code property graphs yield a comprehensive source of information on a given piece of source code. We have chosen to extract paths from these graphs to train our models. Future work might aim to omit this step of extracting paths and research if classifiers based on graph neural networks (GNN) can achieve similar performance based solely off the data and structure in the CPGs.

A

SAMATE PHP Code Sample

```php
<?php
/*
Unsafe sample
input : get the field UserData from the variable $_POST
SANITIZE : use of mysql_real_escape string
construction : interpretation
*/

/*Copyright 2015 Bertrand STIVALET
Permission is hereby granted, without written agreement or royalty fee, to
use, copy, modify, and distribute this software and its documentation for
any purpose, provided that the above copyright notice and the following
three paragraphs appear in all copies of this software.

IN NO EVENT SHALL AUTHORS BE LIABLE TO ANY PARTY FOR DIRECT,
INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF AUTHORS HAVE
BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

AUTHORS SPECIFICALLY DISCLAIM ANY WARRANTIES INCLUDING, BUT NOT
LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE, AND NON-INFRINGEMENT.

THE SOFTWARE IS PROVIDED ON AN "AS-IS" BASIS AND AUTHORS HAVE NO
OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR
MODIFICATIONS.*/


$tainted = $_POST['UserData'];
$tainted = mysql_real_escape_string($tainted);
$query = "SELECT Trim(a.FirstName) & ' ' & Trim(a.LastName) AS employee_name, a.city,
 ↪  a.street & (' ' +a.housenum) AS address FROM Employees AS a  WHERE a.supervisor=
 ↪  $tainted ";

//flaw
$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password'); // Connection to
 ↪  the database (address, user, password)
mysql_select_db('dbname') ;
echo "query : ". $query ."<br /><br />" ;
```

```
37
38 $res = mysql_query($query); //execution
39
40 while($data =mysql_fetch_array($res)){
41 print_r($data) ;
42 echo "<br />" ;
43 }
44 mysql_close($conn);
45 ?>
```

Listing 2: Sample №167182, *CWE_89__POST__func_mysql_real_escape_string__multiple_AS-interpretation.php* from the SAMATE test suite

B

Full Code Property Graph sample №167182

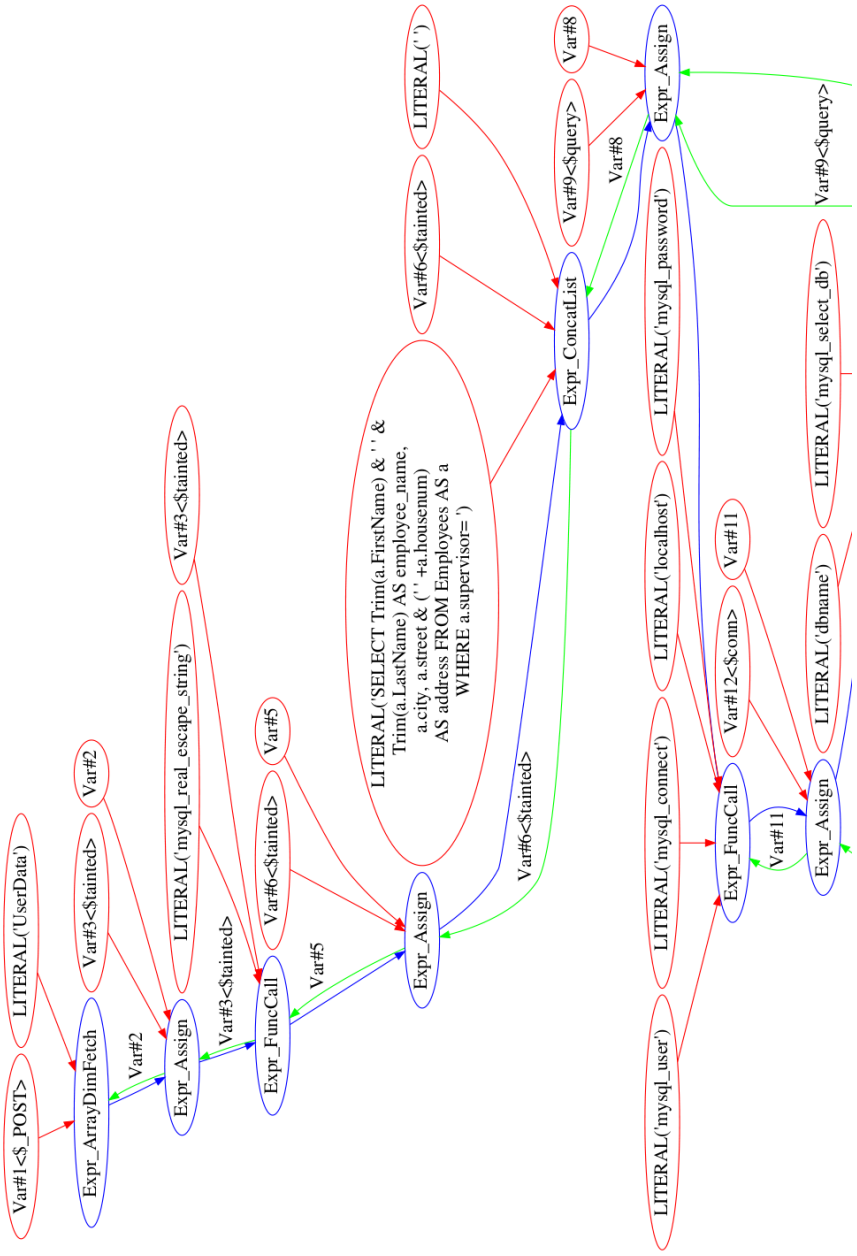
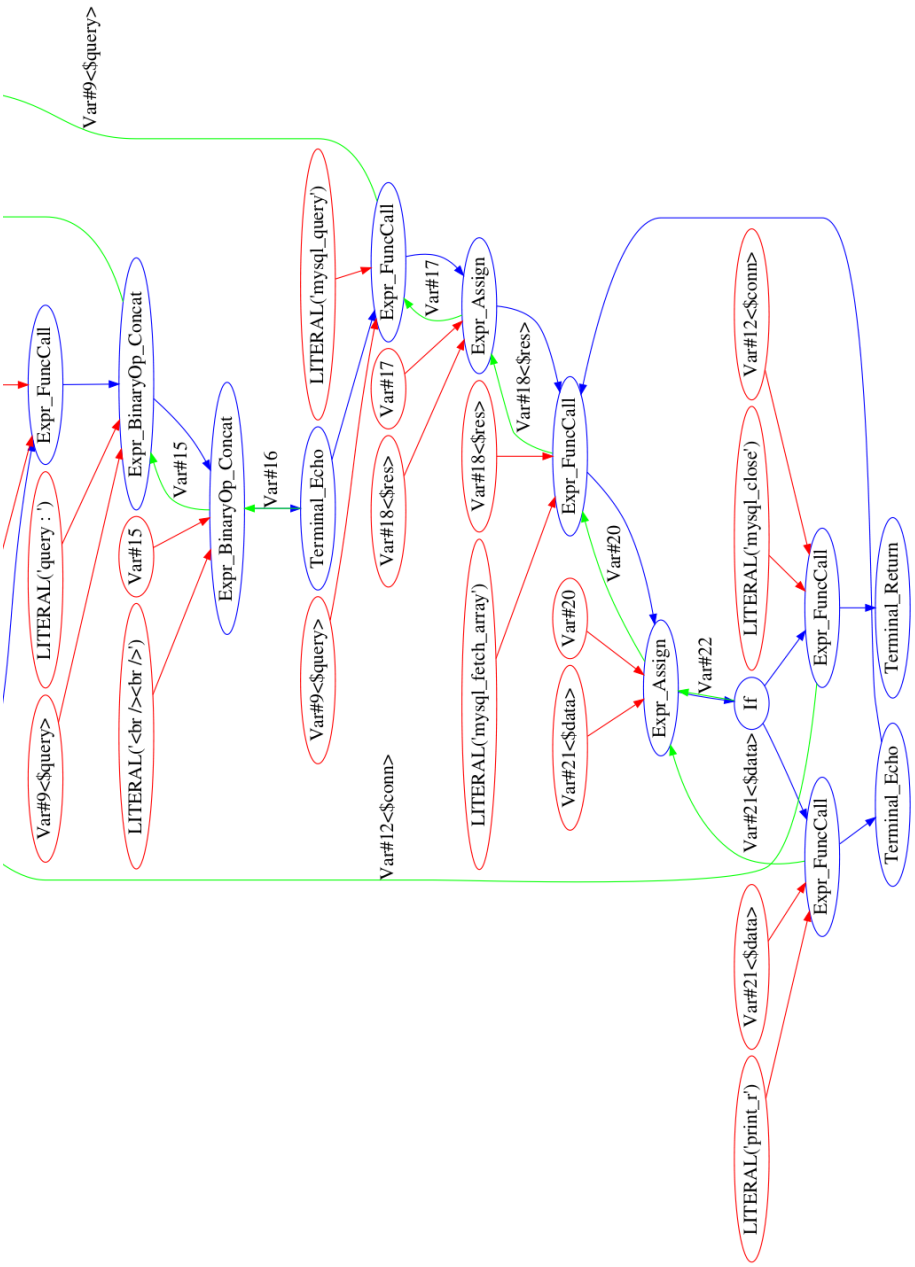


Figure B.1: CPG representation of sample №167182, *CWE_89__POST__func_mysql_real_escape_string__multiple_AS-interpretation.php* from the SAMATE test suite. Blue vertices represent control flow dependencies, green vertices represent a data dependency and red vertices indicate a relation in terms of the abstract syntax tree.



BIBLIOGRAPHY

BOOKS

[18] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[23] W. Richert, *Building machine learning systems with Python*. Packt Publishing Ltd, 2013, p. 142.

[29] J. Patterson and A. Gibson, *Deep learning: A practitioner's approach*. " O'Reilly Media, Inc.", 2017.

[31] F. Chollet, *Deep learning with python*. Manning Publications Co., 2017.

ACADEMIC ARTICLES

[1] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees", in *Proceedings of the 28th Annual Computer Security Applications Conference*, ACM, 2012, pp. 359–368.

[2] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs", in *Security and Privacy (SP), 2014 IEEE Symposium on*, IEEE, 2014, pp. 590–604.

[4] E. M. Hutchins, M. J. Cloppert, and R. M. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains", *Leading Issues in Information Warfare & Security Research*, vol. 1, no. 1, p. 80, 2011.

[7] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey", *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, p. 56, 2017.

[17] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities", in *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, IEEE, 2017, pp. 334–349.

[24] T. Fawcett, "Roc graphs: Notes and practical considerations for researchers", *Machine learning*, vol. 31, no. 1, pp. 1–38, 2004.

[26] T. Saito and M. Rehmsmeier, "The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets", *PloS one*, vol. 10, no. 3, e0118432, 2015.

[33] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting", *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[34] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014.

[35] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection", *arXiv preprint arXiv:1801.01681*, 2018.

[36] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, and J. Wang, "Sysevr: A framework for using deep learning to detect software vulnerabilities", *arXiv preprint arXiv:1807.06756*, 2018.

[37] L. Mou, G. Li, Y. Liu, H. Peng, Z. Jin, Y. Xu, and L. Zhang, "Building program vector representations for deep learning", *arXiv preprint arXiv:1409.3358*, 2014.

[38] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code", *arXiv preprint arXiv:1803.09473*, 2018.

[39] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities", in *Security and Privacy (SP), 2015 IEEE Symposium on*, IEEE, 2015, pp. 797–812.

[40] W. Xiaomeng, Z. Tao, W. Runpu, X. Wei, and H. Changyu, "Cpgva: Code property graph based vulnerability analysis by deep learning", in *2018 10th International Conference on Advanced Infocomm Technology (ICAIT)*, IEEE, 2018, pp. 184–188.

[42] B. Stivalet and E. Fong, "Large scale generation of complex and faulty php test cases", in *2016 IEEE International conference on software testing, verification and validation (ICST)*, IEEE, 2016, pp. 409–415.

[43] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)", 2006.

[44] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction", in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, IEEE, 2016, pp. 297–308.

[45] R. L. Russell, L. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning", *arXiv preprint arXiv:1807.04320*, 2018.

[46] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction", *arXiv preprint arXiv:1708.02368*, 2017.

MISCELLANEOUS

[3] B. E. Strom, J. A. Battaglia, M. S. Kemmerer, W. Kupersanin, D. P. Miller, C. Wampler, S. M. Whitley, and R. D. Wolf, "Finding cyber threats with att&ck-based analytics", Technical Report MTR170202, MITRE, Tech. Rep., 2017.

[5] A. Forni and R. van der Meulen. (2017). Gartner says detection and response is top security priority for organizations in 2017, Gartner, [Online]. Available: https://www.gartner.com/newsroom/id/3638017 (visited on 08/09/2019).

[6] A. Bhutani and P. Wadhwani, *Global cyber security market size worth $300bn by 2024*, 2019. [Online]. Available: https://www.gminsights.com/pressrelease/cyber-security-market.

[8] W3 techs, *Usage statistics and market share of server-side programming languages for websites, november 2018*, [Online; accessed 25-November-2018], 2018. [Online]. Available: https://w3techs.com/technologies/overview/programming_language/all.

[9] R. Shirley. (). Internet security glossary - ietf, [Online]. Available: https://www.ietf.org/rfc/rfc2828.txt.

[10] National Vulnerability Database, Ed. (2019). Nvd - search results for 2019. (visited on 1/12/2019), [Online]. Available: https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&search_type=all&pub_start_date=01%2F01%2F2019&pub_end_date=12%2F31%2F2019.

[11] National Vulnerability Database, Ed. (). Nvd - cwe slice, [Online]. Available: https://nvd.nist.gov/vuln/categories (visited on 09/09/2018).

[12] The OWASP Foundation. (). Cross-site scripting. OWASP, Ed., [Online]. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) (visited on 12/01/2019).

[13] ——, (). Sql injection - owasp. OWASP, Ed., [Online]. Available: https://www.owasp.org/index.php/SQL_Injection (visited on 12/01/2019).

[14] ——, (). Command injection. OWASP, Ed., [Online]. Available: https://owasp.org/www-community/attacks/Command_Injection.

[15] MITRE. (). Open redirect. MITRE, Ed., [Online]. Available: https://cwe.mitre.org/data/definitions/601.html.

[16] J. Kronjee, "Discovering vulnerabilities using data-flow analysis and machine learning.", Master's thesis, Open Universiteit Nederland, Mar. 2018.

[19] M. White and A. White, "Lecture1: Course introduction", in *Fundamentals of Reinforcement Learning*, Coursera Course, 2019. [Online]. Available: https://https://www.coursera.org/learn/fundamentals-of-reinforcement-learning.

[20] Google. (2019). Classification: True vs. false and positive vs. negative, Google, [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/true-false-positive-negative (visited on 08/15/2019).

[21] ——, (2019). Classification: Accuracy, Google, [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/accuracy (visited on 08/15/2019).

[22] ——, (2019). Classification: Precision and recall, Google, [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall (visited on 08/15/2019).

[25] ——, (2019). Classification: Roc curve and auc, Google, [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc (visited on 08/15/2019).

[27] scikit-learn developers. (). sklearn.metrics.precision_recall_fscore_support - scikit-learn 0.22.2 documentation. Scikit-learn, Ed., [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_fscore_support.html.

[28] ——, (). sklearn.metrics.classification_report - scikit-learn 0.22.2 documentation. Scikit-learn, Ed., [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html.

[30] A. Ng, "Lecture: Neural networks basics", in *Neural Networks and Deep Learning*, Coursera Course, 2019. [Online]. Available: https://www.coursera.org/learn/neural-networks-deep-learning?specialization=deep-learning.

[32] ——, "Lecture: Shallow neural networks", in *Neural Networks and Deep Learning*, Coursera Course, 2019. [Online]. Available: https://www.coursera.org/learn/neural-networks-deep-learning?specialization=deep-learning.

[41] National Vulnerability Database, *Nvd - vulnerabilities*, https://nvd.nist.gov/vuln, (Visited on 4/11/2018), 2018.

GLOSSARY

CWE-601 Open redirect, Unvalidated redirects and forwards are possible when a web application accepts untrusted input that could cause the web application to redirect the request to a URL contained within untrusted input. By modifying untrusted URL input to a malicious site, an attacker may successfully launch a phishing scam and steal user credentials[1].

CWE-78 OS command injection, Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application[2].

CWE-79 Cross site scripting, Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it[3].

CWE-89 SQL injection, A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands[4].

[1] Definitions from https://www.owasp.org
[2] ibid
[3] ibid
[4] ibid